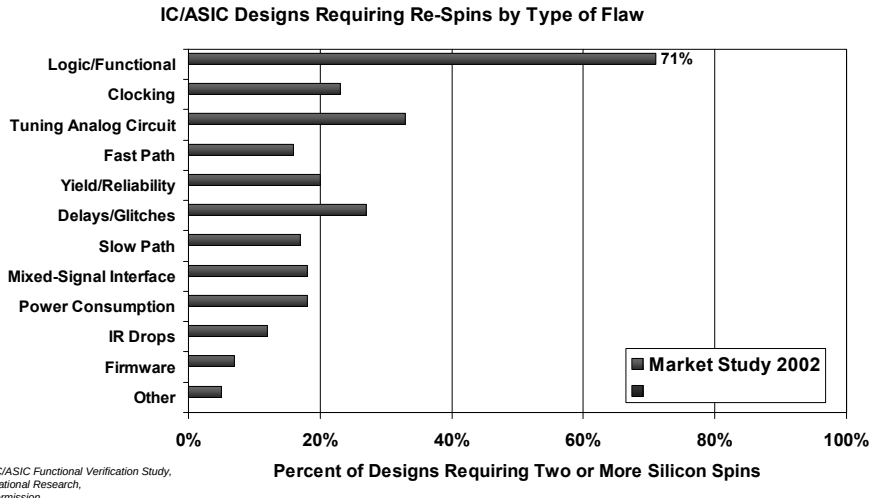
Design Verification Tutorial:
Building Modular, Reusable, Transaction-Level Testbenches in SystemVerilog

Tom Fitzpatrick
Verification Technologist
Mentor Graphics Corp.

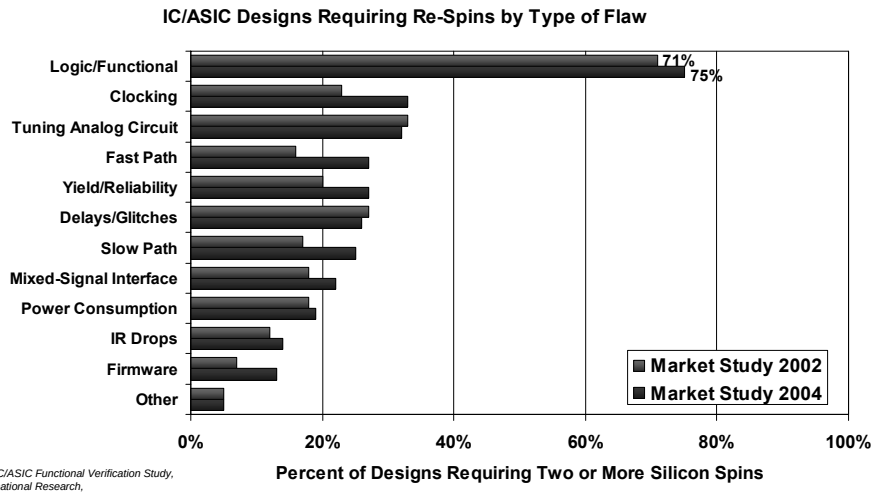
2006 MAPLD International Conference
Washington, D.C.
September 25, 2006

Functional Flaws Driving Need for Re-Spins



Source:
2004/2002 IC/ASIC Functional Verification Study, Collett International Research, Used with Permission

Functional Flaws Driving Need for Re-Spins

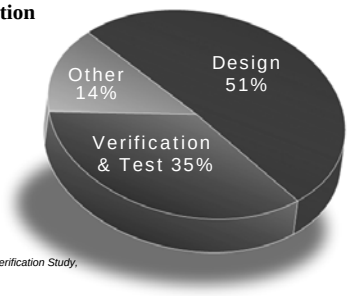


Source:
2004/2002 IC/ASIC Functional Verification Study, Collett International Research, Used with Permission

The Problem is Getting Worse

Design Engineers Are Becoming…

Breakdown of Design Teams by Main Function

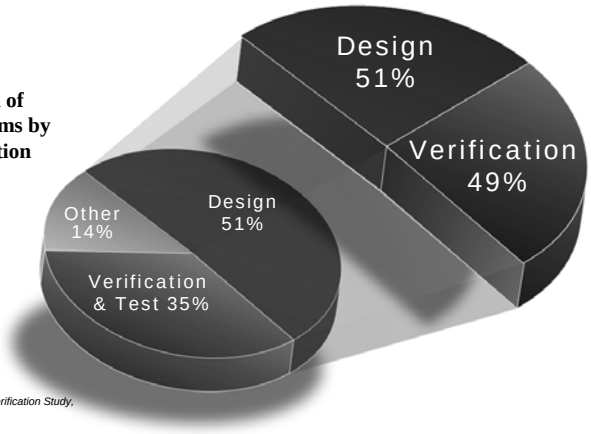


Source:
2004/2002 IC/ASIC Functional Verification Study, Collett International Research, Used with Permission

Design Engineers Are Becoming…

Verification Engineers

Breakdown of Design Teams by Main Function



Source:
2004/2002 IC/ASIC Functional Verification Study,
Collett International Research,
Used with Permission

You've Seen a Technology Explosion Targeting Verification

Energy from a black hole NCG 4696

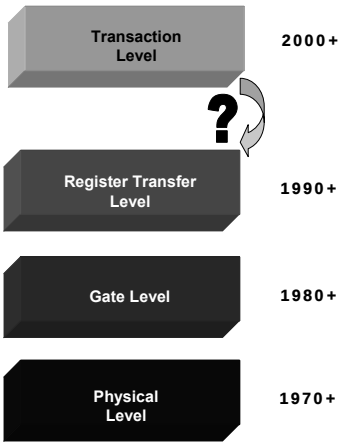
- Assertion-based verification
- Functional coverage
- Constrained-random testing
- Coverage-driven verification
- Dynamic-formal verification
- Transaction-level verification
- Model checking
- And more . . .

Complexity and Abstraction

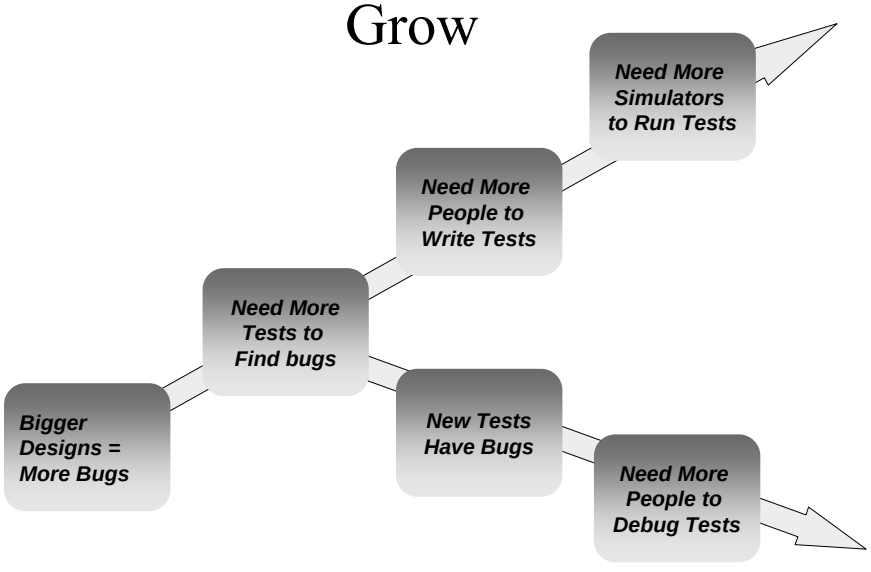
- Moore's Law drives higher levels of abstraction

```
always @(decade)
  abstraction.raise();
```

- Currently shifting from RTL to TLM
  - No automated refinement path
  - Need methodology to assist refinement

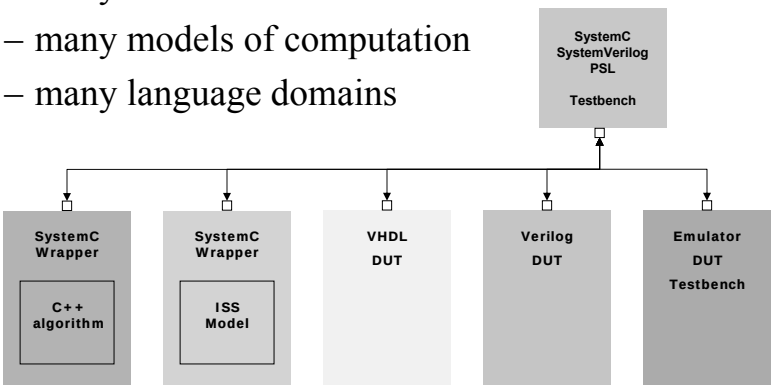


Verification Problem Continues To Grow

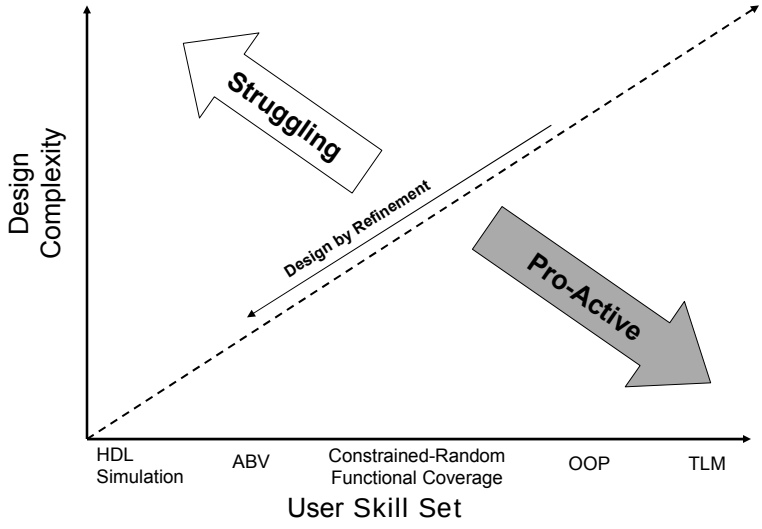


Mix & Match: Languages & Engines

- Typically, in one simulation, we integrate
  - many abstraction levels
  - many models of computation
  - many language domains

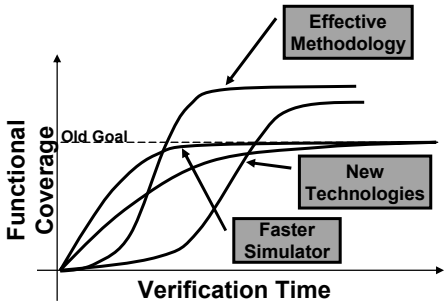


Complexity-Based Requirements



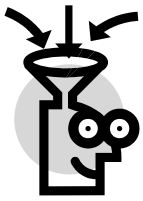
Methodology Basics

- People used to think all they needed for better verification was a faster simulator
- Then they thought they needed new technologies
  - Testbench
  - Assertions
- Now they realize they need to know how to use them effectively
  - This is "Methodology"



What is "Methodology"?

- The study of methods
- A body of practices, procedures, and rules used by those who work in a discipline



- Knowledge of what works and what doesn't
  - Standards enable communication

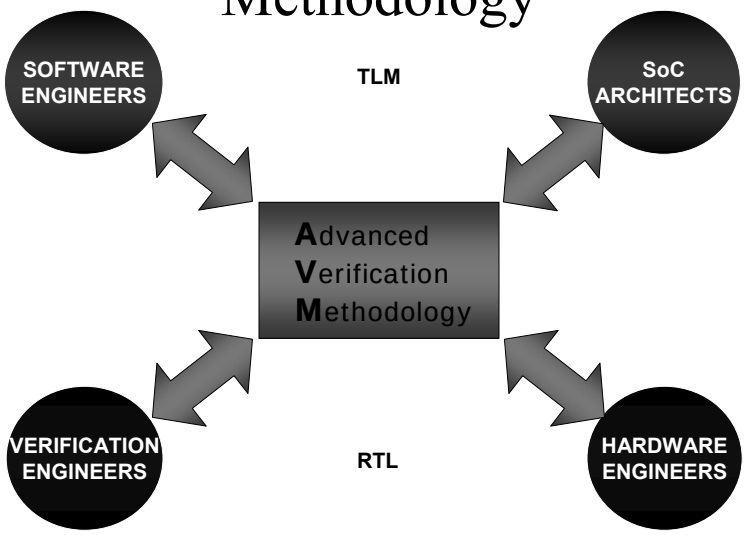
Methodology Lets You Get Information

- If a tree falls in the forest and no one hears it, it does NOT make a sound
- You have to be paying attention
  - Assertions
  - Functional Coverage
  - Scoreboarding
- You have to knock down the right trees
  - Generate interesting stimulus
  - Use the right tools
    - Directed stimulus
    - Constrained-Random Stimulus
    - Formal verification

Verification is a Human Problem Too

Misinterpreting specifications is a prime source of bugs

Mentor Advanced Verification Methodology

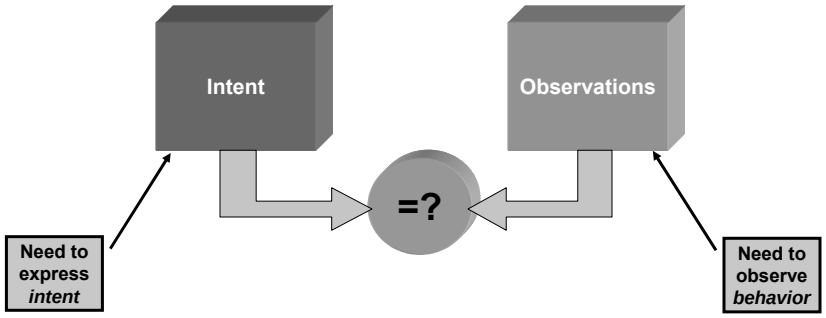


Key Aspects of a Good Methodology

- Automation
  - Let the tools do the work
- Observability
  - Self-checking is critical
  - Automate self-checking and coverage tracking
  - Assertion-based Verification
- Controllability
  - Make sure you exercise critical functionality
  - Constrained-Random stimulus and formal verification automate the control process
- Verification Planning and coordination
- Reusability
  - Don't reinvent the wheel
  - Reusability is functionality- and/or protocol-specific
  - Reuse is critical across projects, across teams, and across tools
- Measurability
  - If you can't measure it, you can't improve it
  - Tools automate the collection of information
  - Analysis requires tools to provide information in useful ways
  - All tools must consistently contribute to measurement and analysis

Verifying a Design means…

- Showing that a system's behavior matches its designer's intent



Design vs. Verification

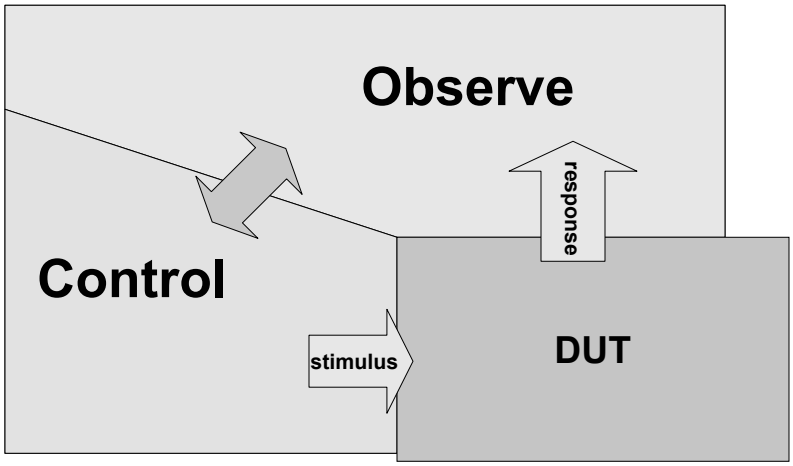
Verification:

- A *software model* of the environment
- Expresses the *intended behavior* of the design
  - Assertions describe intent declaratively
  - Testbench response checkers and golden models describe intent procedurally
- Gathers *information* to measure progress (i.e. coverage)
- Must interact with design at different levels of abstraction

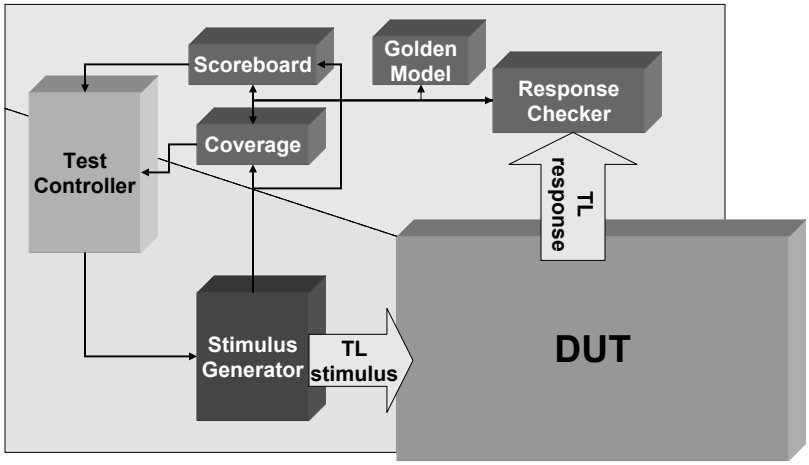
Design:

- Describes Hardware
- Multiple Abstraction Levels
  - System, RTL, gate

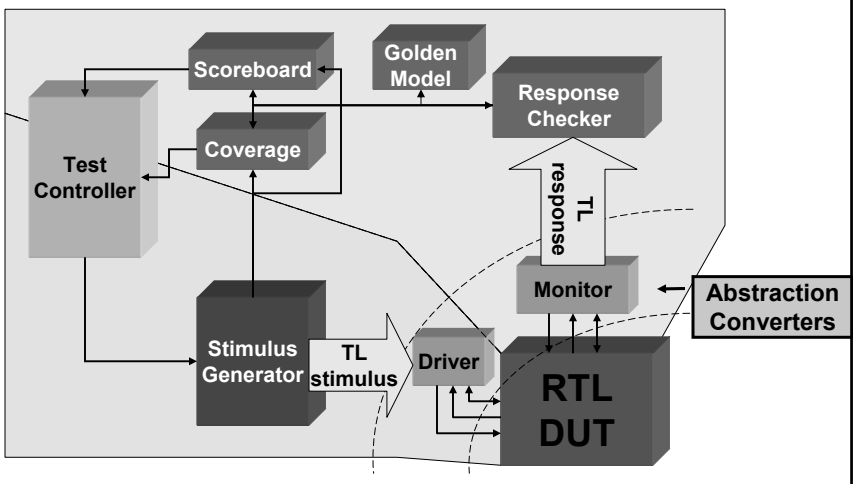
Deconstructing Verification



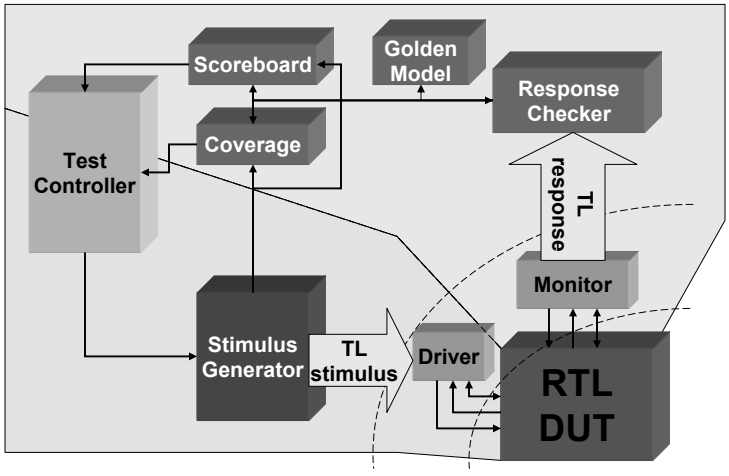
Simulation Verification: The Testbench



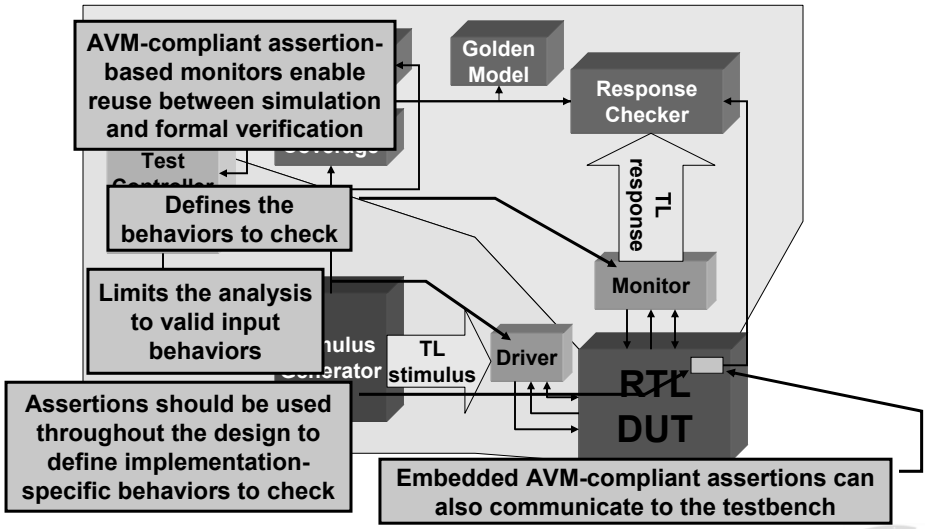
Refinement



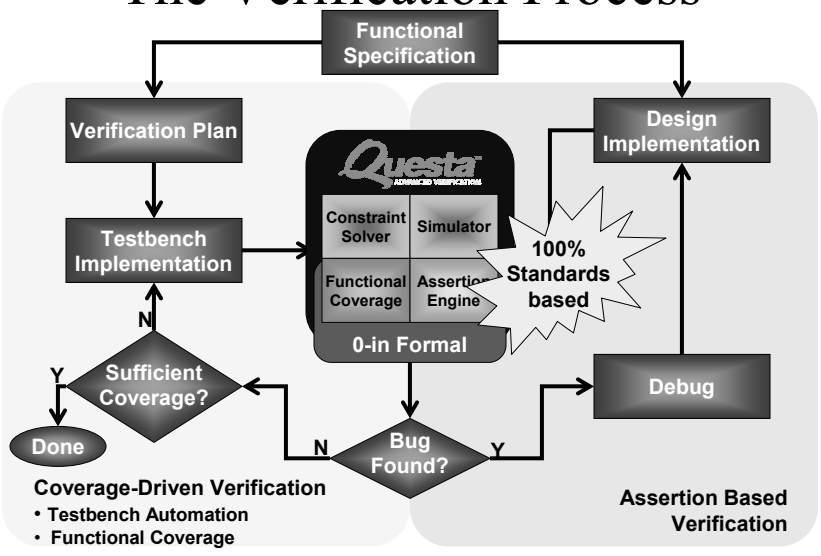
Reuse



Formal Verification: Assertions



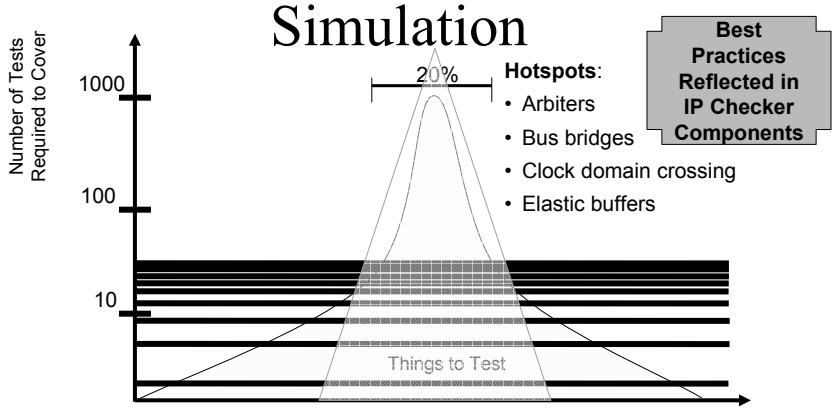
The Verification Process



The Verification Plan

- The Verification Plan is a list of questions you need to answer about your design
- What *information* do you need to answer the questions?
  - Does the design do everything it's supposed to do?
    - Does it work (whatever "work" means)? Is it fast enough?
    - When X happens, will it do Y?
    - Do all that goes in - come out when they're supposed to?
    - Does it fully support all bus protocols it's supposed to?
  - Does the design do anything it's *not* supposed to do?
    - Will it recover gracefully if something unexpected happens?
    - Can I be sure it will only do Y when X happens?
  - When have I successfully answered the first two questions?

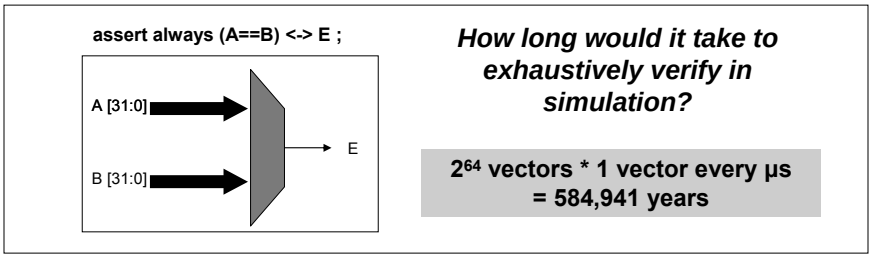
Targeted Methods Complement Simulation



- ABV and exhaustive formal verification
  - Complete verification of hotspots
  - Focus formal verification for largest return-on-investment
  - Replaces millions of cycles of simulation

Simulation May Not Be Enough

*Some blocks are so important they **must** be verified exhaustively*

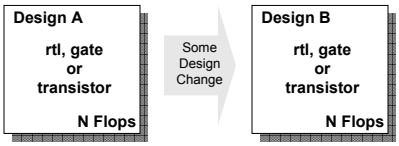


- Formal and simulation complement each other to provide
  - Exhaustive verification
  - Advanced bug finding
  - Coverage improvement
  - Interface compliance

Equivalence vs. Property Checking

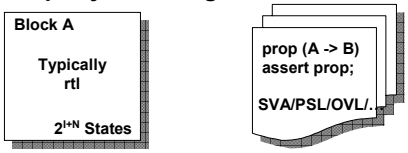
- Design A = Design B?
  - Chip level runs
  - Specific Algorithm
  - Explores equivalency of cones of logic between states
- Assertion = Design Behavior?
  - Block level runs
  - Many Algorithms
  - Exhaustive state space exploration of the block

Equivalency Checking



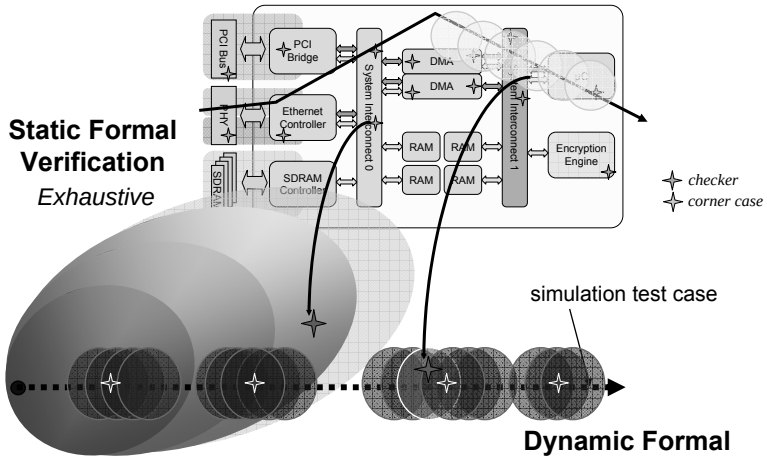
Pass: Design A = Design B
Fail: Design A != Design B
Debug shows cones of logic that aren't equal

Property Checking



Proof: Formal Proof for all input conditions
Firing: Some condition violates property
Debug gives counter example showing firing

Formal Verification



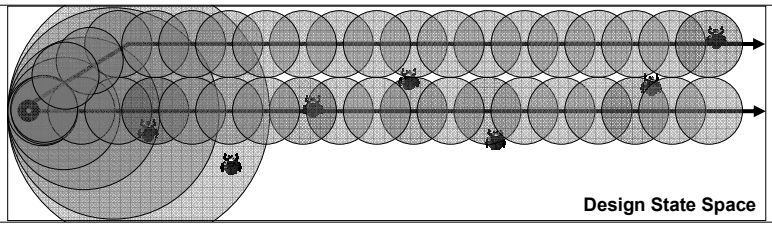
Static Formal Verification
Exhaustive

Dynamic Formal Verification
Locally exhaustive

Powerful formal verification that goes broad **and** deep

How to Fill In Coverage Holes

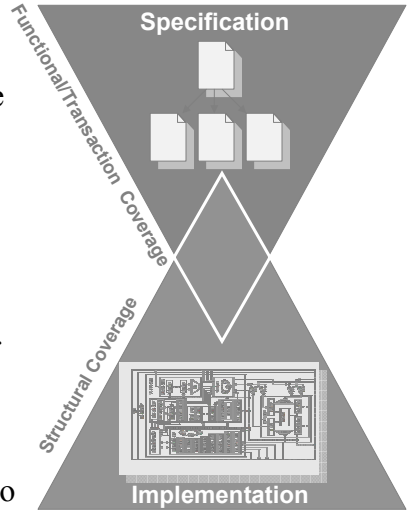
Combine formal with simulation to expand coverage



- Constrained-random testing could miss coverage points
- Constructing directed tests to hit deep states infeasible
- Depth and complexity of coverage points exceed capacity of static formal
- Solution: Combine formal and simulation
  - Formal verification from specific simulation states

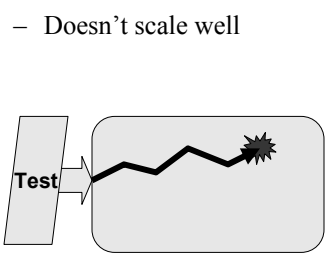
Total Coverage Model

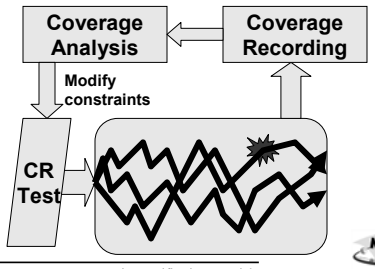
- Functional (specification-based)
  - Checks that all functions of the design are tested
  - Created by verification team
  - Makes sure the design does everything it's supposed to do
- Structural (implementation-based)
  - Checks that all corner-cases of the design are tested
  - Created by designers
  - Makes sure the design doesn't do anything it's not supposed to do



Functional Coverage and Testbench Automation

- In a directed test, the coverage points are coded in the test itself
  - Test writer must code *each* specific scenario to specify intent explicitly
  - Must be able to predict interesting cases in order to code them
  - Doesn't scale well
- With a random test, scenarios cannot be predicted.
  - Need to track information about what happened
  - Support queries to determine if targets were hit
  - *Intent* captured by self-checking/ scoreboard

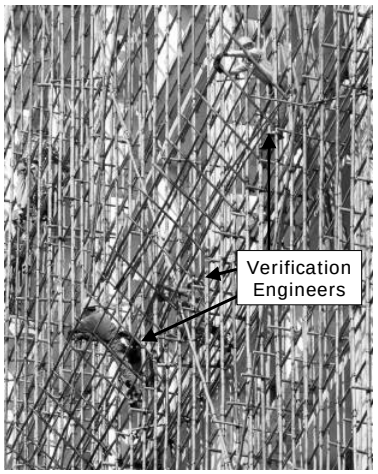




TBA is All About Infrastructure

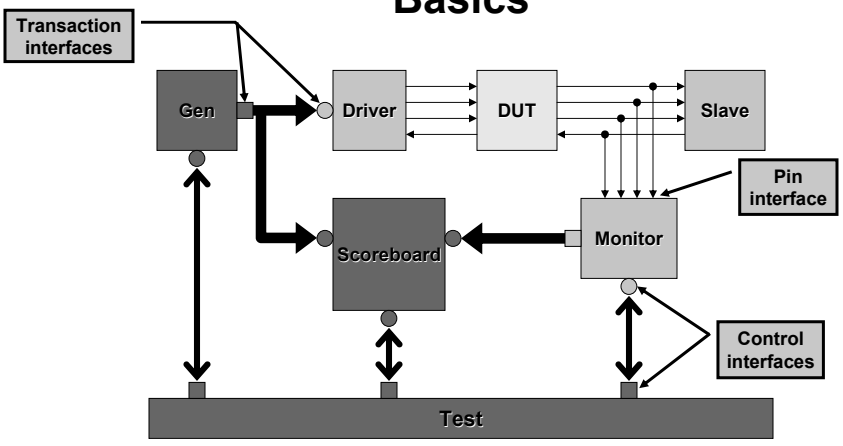
III IV
I II

- Scaffolding around the design
- All the stuff you need to build and verify a design
  - Software
  - Programming-centric view of the *intent*
- Specified in the verification plan
- Can cost as much or more than the design itself
- Efficiency, Reuse, etc are important
- Must be built from the ground-up



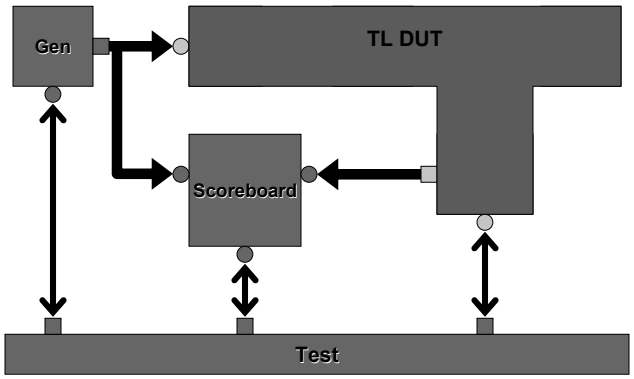
Building a Testbench

Basics



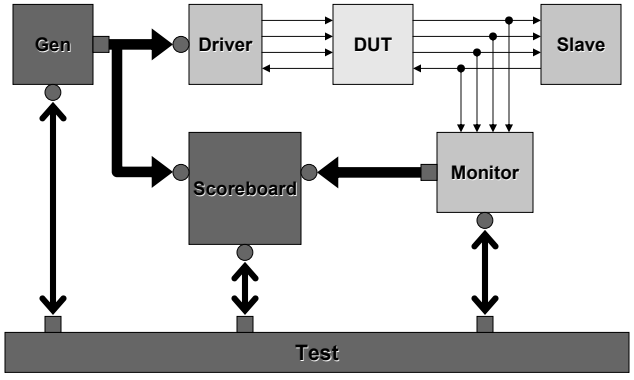
Intent is captured in test and scoreboard

Building a Testbench
Modularity



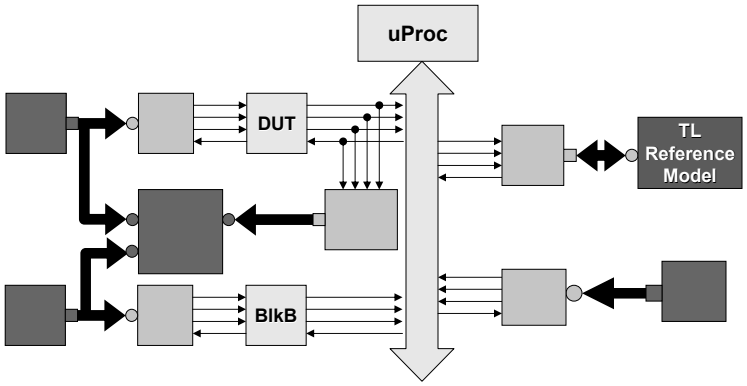
Intent is captured in test and scoreboard

Testbench "Plumbing"
Reuse



Testbench "Plumbing"

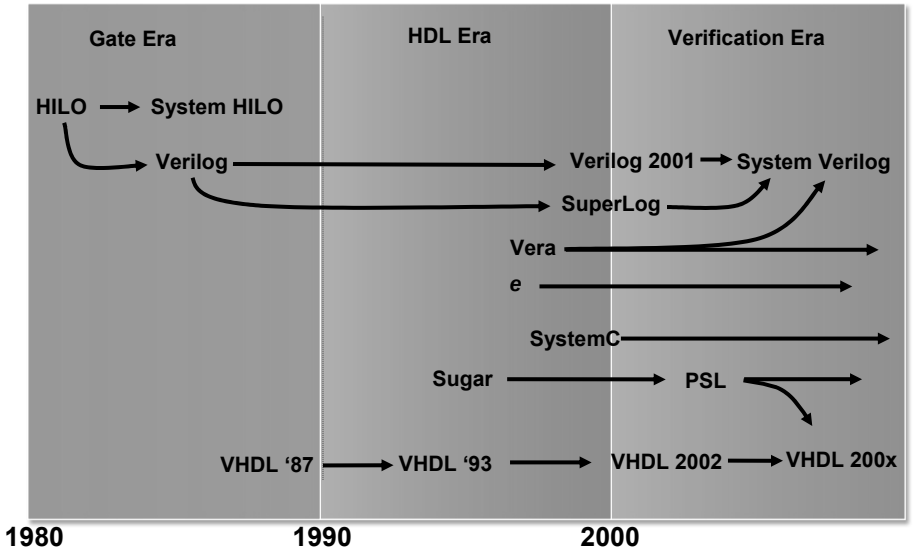
Reuse



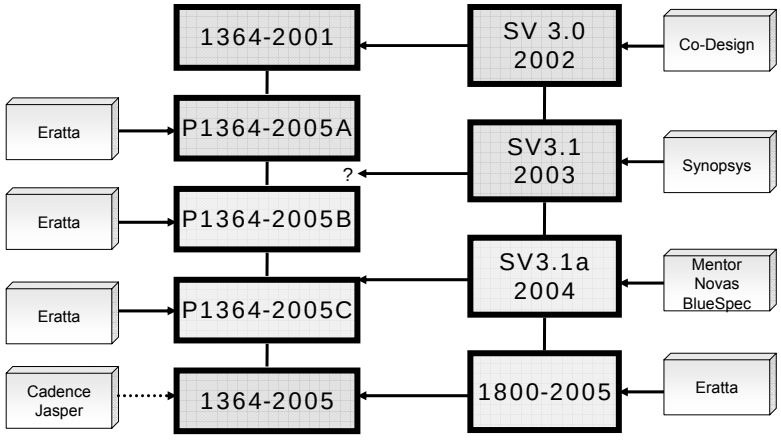
Regular interfaces allow for mix-n-match

SystemVerilog Overview

Language Explosion

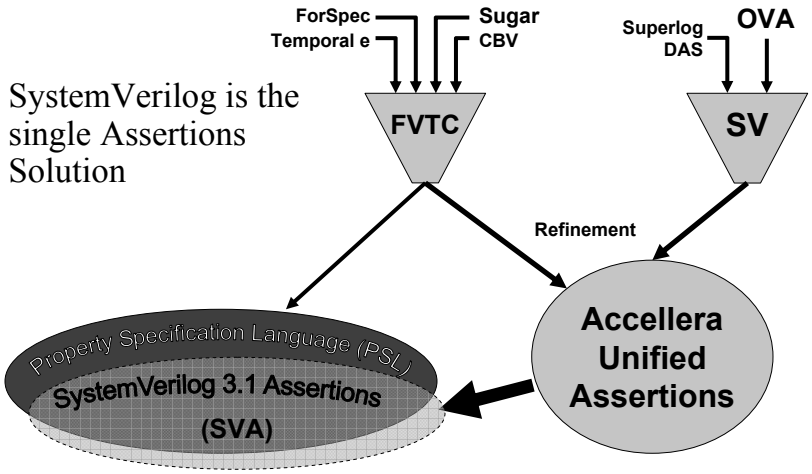


1364 and P1800



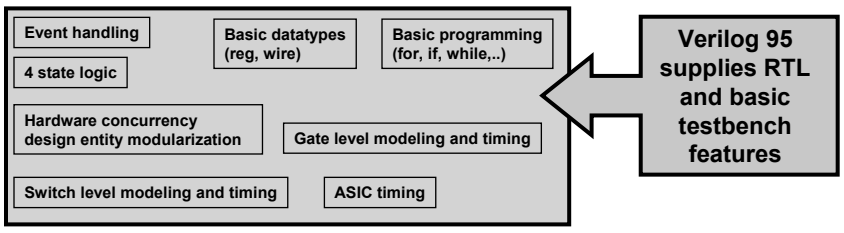
The Accellera Assertions Process

- SystemVerilog is the single Assertions Solution

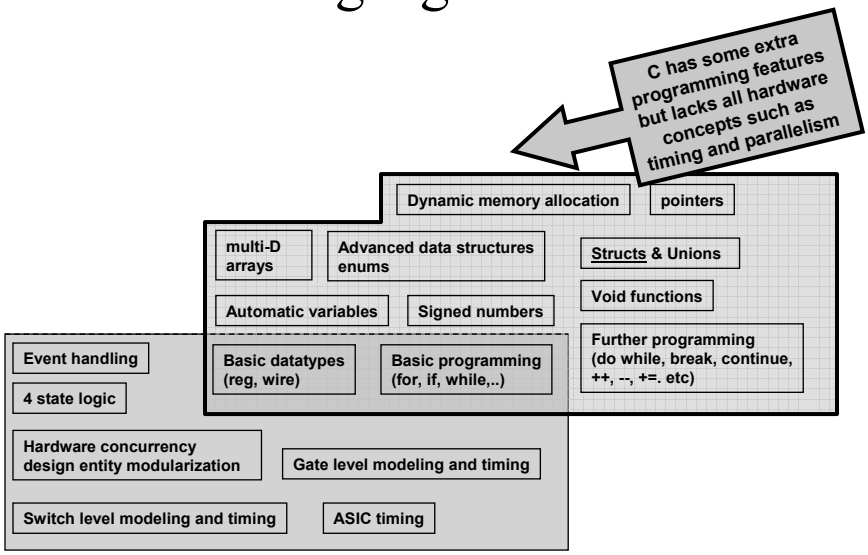


SystemVerilog Semantics

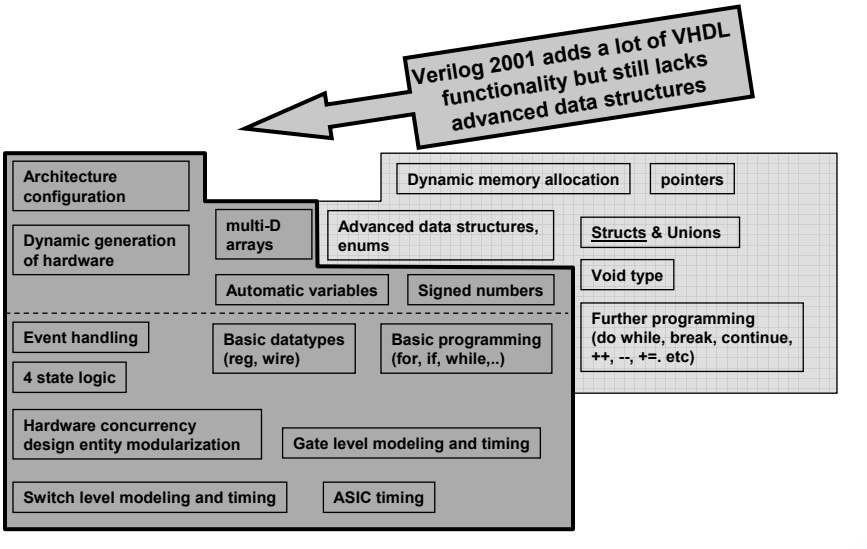
What is the origin of the IEEE 1800 SystemVerilog Semantics?



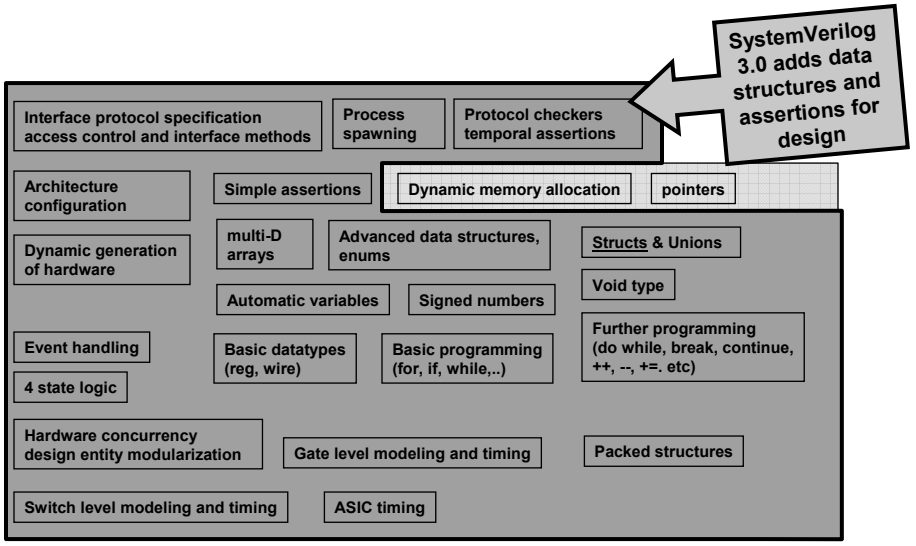
C Language Semantics



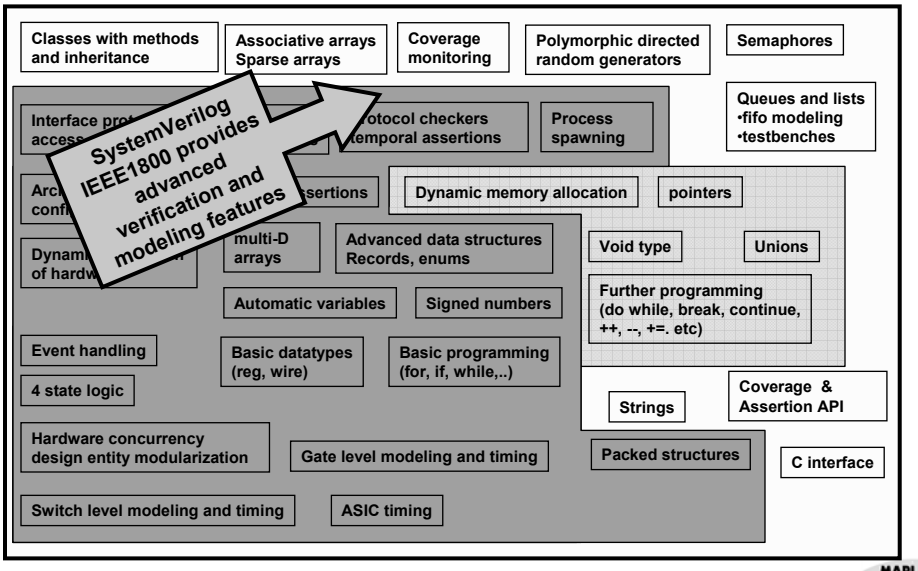
Verilog 2001 Semantics



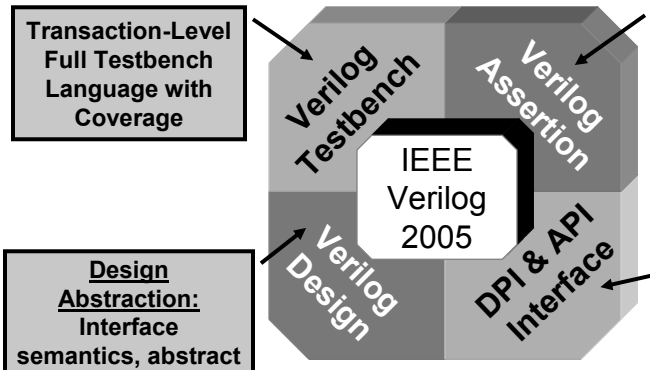
SV 3.0 Semantics



SV 1800 Semantics



SystemVerilog Components



Transaction-Level Full Testbench Language with Coverage

Advanced verification capability for semiformal and formal methods. The Assertion Language Standard for Verilog

Design Abstraction: Interface semantics, abstract data types, abstract operators and expressions

Direct C interface, Assertion API and Coverage API

Defined by Accellera. Completed and standardized under the IEEE

SystemVerilog Technical Committees

IEEE P1800 Structure

Structure correlates to SystemVerilog components

Chair - Johny Srouji

IEEE P1800 SystemVerilog WG

Vice chair – Shrenik Mehta

Secretary - Dennis Brophy

Chair: Karen Pieper

P1800 Errata Sub-WG

Champions

IEEE 1364 Features are now driven by the IEEE 1800 technical committees: Language issues in SV-BC C-Interface issues in SV-CC

SV-BC (*design*)

SV-EC (*testbench*)

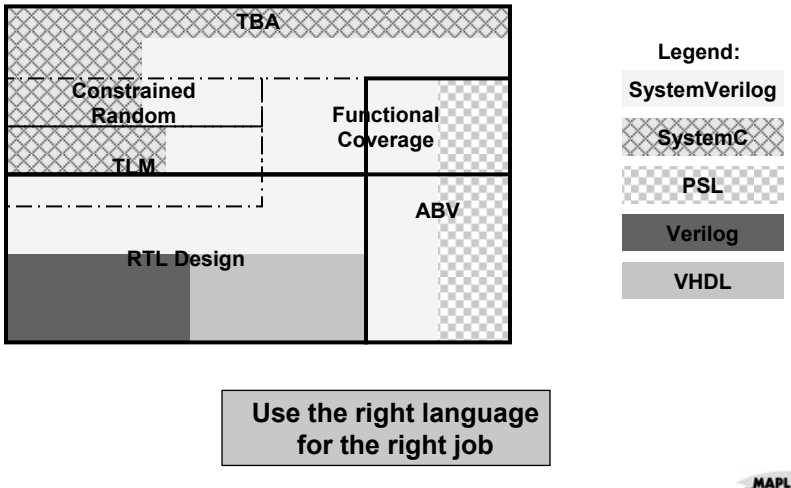
SV-AC (*assertions*)

SV-CC (*DPI*)

Methodology is Not Language

- Languages and tools must *support* a methodology
  - The language does not *define* the methodology
- The Methodology ties different technologies together effectively
  - Assertion-Based Verification (ABV)
  - Testbench Automation (TBA)
    - Constrained-Random Verification (CRV)
  - Coverage-Driven Verification (CDV)
    - Functional Coverage
    - Code Coverage
  - Transaction-Level Modeling (TLM)

Methodology/Language Matrix

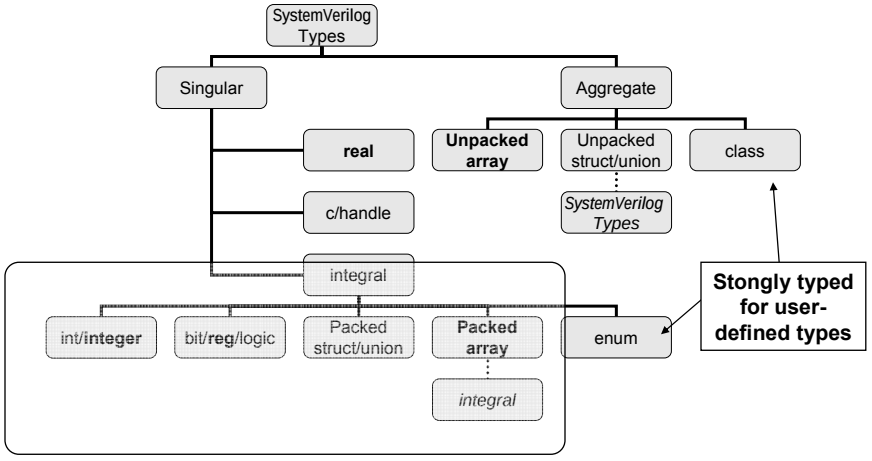


Use the right language for the right job

Broad Industry Support for SystemVerilog

SystemVerilog Design Constructs

SystemVerilog Type System



2 State and 4 State Data Types

Verilog
SystemVerilog

```
reg a;
integer i;
```

Verilog reg and integer type bits can contain x and z values

SystemVerilog

```
logic a;
logic signed [31:0] i;
```

Equivalent to these 4-valued SystemVerilog types

SystemVerilog

```
bit a;
int i;
```

These SystemVerilog types have two-valued bits (0 and 1)

If you don't need the X and Z values then use the SystemVerilog bit and int types which MAKE EXECUTION FASTER

Easing the reg / wire Duality

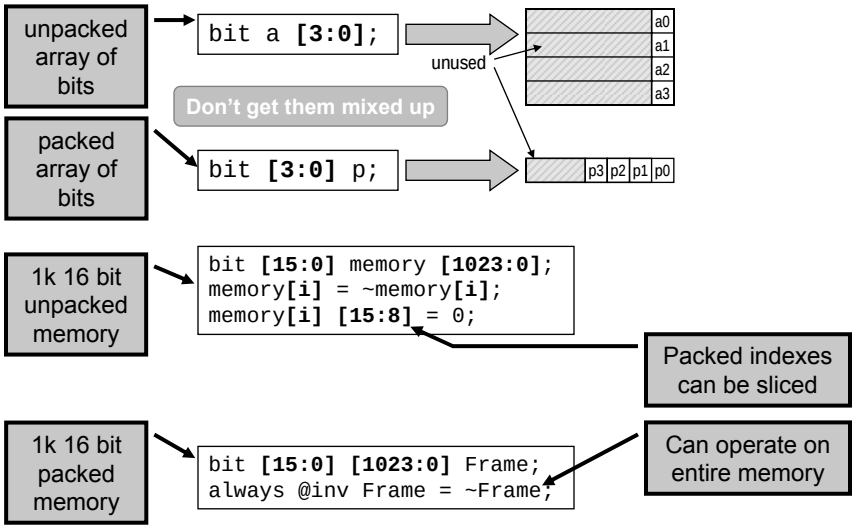
- Moving from RTL to an instance based description
- Replacing regs with wires
- Bit, logic, and reg types can be assigned as regs or driven by a single instantiation

	Verilog	SystemVerilog

RTL	`reg o;`
	`always @(a or b)`
	`  o = a & b;`
		`reg o;`
		`always_comb`
		`  o = a & b;`

Gate	`wire o;` `and aa (o, a, b);`	`reg o;` `and aa (o, a, b);`

Packed And Unpacked Arrays



Structures

```
struct { bit [7:0]     opcode;
         bit [23:0]    addr;
       } IR;           // anonymous structure
```

Like in C but without the optional structure tags before the {

```
typedef struct { bit [7:0]    opcode;
                 bit [23:0]   addr;
} instruction;          // named structure type

instruction IR;         // define variable

IR.opcode = 1;          // set field in IR
```


Unions

```
typedef union {
   int n;
   real f;
   } u_type;

u_type u;

initial
  begin
    u.n = 27;
    $display("n=%d", u.n);
    u.f = 3.1415;
    $display("f=%f",u.f);
    $finish(0);
  end
```

union

provide storage for either **int** or **real**

int

real

again, like in C

structs and unions can be assigned as a whole

Can be passed through tasks/functions/ports as a whole

can contain fixed size packed or unpacked arrays

Packed Structures

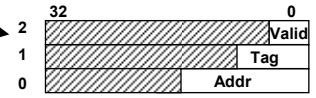
Represents bit or part selects of vectors

```
struct packed {
  bit        Valid;
  byte       Tag;
  bit [15:0] Addr;
} Entry;
iTag   = Entry.Tag;
iAddr  = Entry.Addr;
iValid = Entry.Valid
```

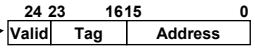
```
reg [24:0] Entry;
`define Valid 24
`define Tag 23:16
`define Addr 15:0
iTag   = Entry[`Tag];
iAddr  = Entry[`Addr];
iValid = Entry[`Valid]
```

packed struct may contain other packed structs or packed arrays

unpacked struct



packed struct



Packed Unions

- Many views for same data

```
typedef union packed {
  ppacket_t f;
  bit_t [11:0][7:0] qbyte;
} upacket_t;
```

src_addr				dst_addr				payload			
11	10	9	8	7	6	5	4	3	2	1	0
95:0											

```
upacket_t upkt;
always_comb begin
  upkt.f.src_adr = node_id;
  upkt.qbyte[7:4] = packet.dst_addr;
  upkt[31:0] = packet.payload;
end
```

Enumerated Data Types

```
enum {red, yellow, green} light1, light2;
```
anonymous int type

```
enum {bronze=3, silver, gold} medal;
```
silver=4, gold=5

```
enum {a=0, b=7, c, d=8} alphabet;
```
Syntax error

```
enum {bronze=4'h3, silver, gold} medal;
```
silver=4'h4, gold=4'h5

```
typedef enum {red, green, blue, yellow, white, black} Colors;

Colors  col;
integer  a, b;

a = blue * 3;
col = yellow;
b = col + green;
```
a=2*3=6
col=3
b=3+1=4

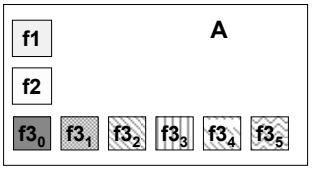
Type Casting

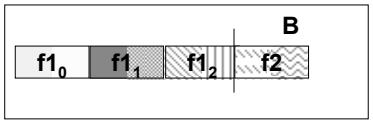
```
int'(2.0 * 3.0)
shortint'{8'hFA, 8'hCE}
17 '(x - 2)
```
A data type can be changed by using a cast (') operation

- Any aggregate bit-level object can be reshaped
  - Packed ⇔ Unpacked, Array ⇔ Structure

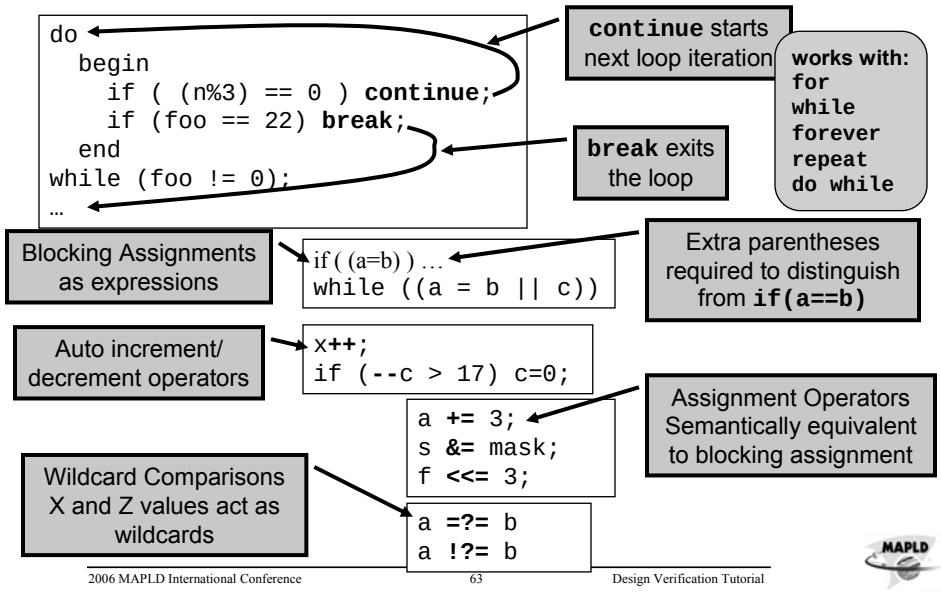
Objects must have identical bit size

```
typedef struct {
        bit [7:0] f1;
        bit [7:0] f2;
        bit [7:0] f3[0:5];
} Unpacked_s;
typedef struct packed {
        bit [15:0][0:2] f1;
        bit [15:0] f2;
} Packed_s;
Unpacked_s A;
Packed_s B;
…
        A = Unpacked_s'(B);
        B = Packed_s'(A);
```





Familiar C Features In SystemVerilog



Re-Use

- Packages facilitate sharing of parameters, data types & functions
- Facilitate better organization and reference to global collateral
- Prevent collision of names in global scope

```
package Protocol;
  parameter ADDR_SIZE = 32;
  parameter DATA_SIZE = 32;
  typedef bit_t [ADDR_SIZE-1:0]
      addr_t;
  typedef bit_t [DATA_SIZE-1:0]
      data_t;
  typedef struct {
 addr_t src_addr;
 addr_t dst_addr
 data_t payload;
  } packet_t;
  function automatic bit_t
EvenParity32 (bit_t [31:0] i);
  return ^i;
  endfunction
endpackage
```

```
bit_t [Protocol::ADDR_SIZE-1:0]
        tmp_address;
…
import Protocol::*;
bit_t [ADDR_SIZE-1:0] tmp_address;
```

```
bit_t [Protocol::ADDR_SIZE-1:0] node_adr;
bit_t [Memory::ADDR_SIZE-1:0]  dram_adr;
import Protocol::*;
```

Re-Use

- Data Type parameter extends generic capabilities
  - For modules

```
module dff #(parameter type T = bit_t)
            (output T q,
             input T d,
             input ck);
  always_ff @(posedge ck)
    q <= d;
endmodule
```

```
packet_t i_pkt,o_pkt;
bit_t ck;
dff #(.T(packet_t)) ff0
(.q(o_pkt),.d(i_pkt),.ck(ck));
```

- A must when using unpacked structs & arrays
- Useful for creating generic code

```
localparam type T = type(i_pkt);
T tmpfifo[7:0];
always_comb
      tmpfifo[0] = i_pkt;
```

Type operator alleviates need for referring to data type by name

Macros

- SystemVerilog improves macros
  - Insert macro argument into a string

```
`define ATTR(arg) (*myattr=`"arg`"*)
  Source: `ATTR(decl) bit_t flag;
Expanded: (*attr="decl"*) bit_t flag;
```

  - Create identifiers

```
`define IDENT(arg) inst_``arg
  Source: dff #(.T(t_bit))`IDENT(o)(.q(q),.d(d),.clock(clock));
Expanded: dff #(.T(t_bit)) inst_o (.q(q),.d(d),.clock(clock));
```

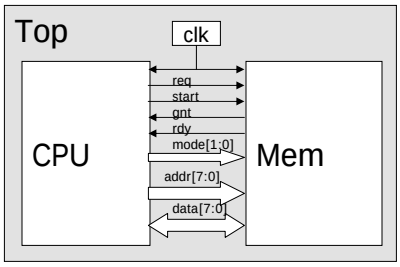
- Extends utility of macros to save typing and promote uniformity

SystemVerilog Interfaces

Interconnect: The Old Way

```
module memMod(input  logic req,
                     bit clk,
                     logic start,
                     logic[1:0] mode,
                     logic[7:0] addr,
              inout  logic[7:0] data,
              output logic gnt,
                     logic rdy);
always @(posedge clk)
  gnt <= req & avail;
endmodule

module cpuMod(input  bit clk,
                     logic gnt,
                     logic rdy,
              inout  logic [7:0] data,
              output logic req,
                     logic start,
                     logic[7:0] addr,
                     logic[1:0] mode);
endmodule
```

```
module top;
  logic req,gnt,start,rdy;
  bit   clk = 0;
  logic [1:0] mode;
  logic [7:0] addr,data;

memMod mem(req,clk,start,mode,
           addr,data,gnt,rdy);
cpuMod cpu(clk,gnt,rdy,data,
           req,start,addr,mode);
endmodule
```



Interconnect: Using Interfaces

```
interface simple_bus;
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;
endinterface: simple_bus

module memMod(interface a,
              input bit clk);
  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule

module cpuMod(interface b,
              input bit clk);
endmodule
```

Bundle signals in interface

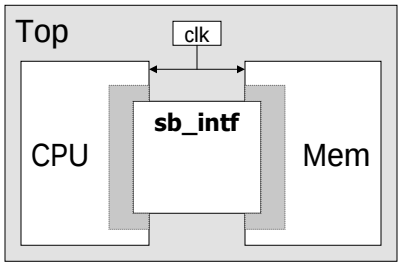
Use interface keyword in port list

Refer to intf signals

```
module top;
  bit clk = 0;
  simple_bus sb_intf;

  memMod mem(sb_intf, clk);
  cpuMod cpu(.b(sb_intf),
             .clk(clk));
endmodule
```

interface instance

Connect interface



Encapsulating Communication

Parallel Interface

```
interface parallel(input bit clk);

  logic [31:0] data_bus;
  logic data_valid=0;

task write(input data_type d);
    data_bus <= d;
    data_valid <= 1;
    @(posedge clk) data_bus <= 'z;
    data_valid <= 0;
  endtask

task read(output data_type d);
    while (data_valid !== 1)
                @(posedge clk);
    d = data_bus;
    @(posedge clk) ;
  endtask

endinterface
```

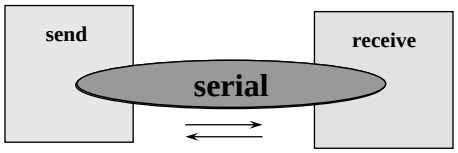
Serial Interface

```
interface serial(input bit clk);
  logic data_wire;
  logic data_start=0;

  task write(input data_type d);
    for (int i = 0; i <= 31; i++)
      @(posedge clk) begin
        if (i==0)  data_start <= 1;
          else  data_start <= 0;
        data_wire = d[i];
      end
  endtask

  task read(output data_type d);
    while (data_start !== 1)
      @(negedge clk);
    for (int i = 0; i <= 31; i++)
      @(negedge clk)
        d[i] <= data_wire;
  endtask
endinterface
```

Using Different Interfaces

```
typedef logic [31:0]
data_type;

bit clk;
always #100 clk = !clk;

parallel channel(clk);
send    s(clk, channel);
receive r(clk, channel);
```

```
typedef logic [31:0]
data_type;

bit clk;
always #100 clk = !clk;

serial channel(clk);
send    s(clk, channel);
receive r(clk, channel);
```




```
module send(input bit clk,
            interface i);
  data_type d;
...
  i.write(d);
endmodule
```

Module inherits communication method from interface

Using Tasks in an Interface

```
interface simple_bus(input bit clk);
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;
  task masterRd(input logic[7:0] raddr);
    ...
  endtask:masterRd

  task slaveRd;
    ...
  endtask:slaveRd
endinterface: simple_bus

module memMod(interface a);
  ...
 always @(a.start)
    a.slaveRd;
endmodule

module cpuMod(interface b);
endmodule
```

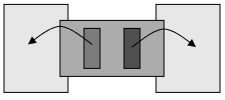
Communication Task Encapsulated in Interface

Interface Method Called From Intantiating Module

```
module top;
  bit clk = 0;
  simple_bus sb_intf(clk);

  memMod mem(sb_intf);
  cpuMod cpu(.b(sb_intf));
endmodule
```

By default, all Interface Methods can be called from any module that instantiates the Interface

What if we want to restrict "slave" modules from calling the masterRd task?



Modports: Importing Tasks From an Interface

```
interface simple_bus(input bit clk);
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;

  modport slave(input req,addr,mode,start,clk,
                output gnt,rdy,
                inout data,
                import task slaveRd(),
                       task slaveWr());

  modport initiator(input gnt,rdy,clk,
                    output req,addr,
                           mode,start,
                    inout data,
                    import task masterRd(input logic[7:0] raddr),
                           task masterWr(input logic[7:0] waddr));
...
endinterface
```

Import into module that uses the modport

Modules using *initiator* modport can only call these tasks

Modports: Using Imported Tasks

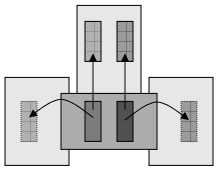
```
module memMod(interface.slave a);
logic avail;

  always @(posedge a.clk)
    a.gnt <= a.req & avail;

  always @(a.start)
    if(a.mode[0] == 1'b0)
      a.slaveRead;
    else
      a.slaveWrite;
endmodule

module cpuMod(interface b);
  enum {read,write} instr;
  logic [7:0] raddr;
...
  always @(posedge b.clk)
    if(instr == read)
      b.masterRead(raddr);
       ...
    else
      b.masterWrite(raddr);
endmodule
```

Only has access to slaveRd/slaveWr tasks

```
module omniMod(interface b);
   //...
endmodule:omniMod

module top;
  logic clk = 0;

  simple_bus sb_intf(clk);

  memMod mem(sb_intf);
  cpuMod cpu(sb_intf.initiator);
  omniMod omni(sb_intf);
endmodule
```

Only has access to masterRd/Wr tasks

Has access to all tasks and signals



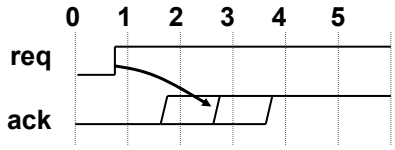
Tasks & Functions in Interfaces

- Allows More Abstract Modeling
  - Transaction can be executed by calling a task without referring to specific signals
  - "Master" module can just call the tasks
- Modports Control Sharing of Methods
  - Methods defined outside a module are "imported" into a module via modport
  - Effectively gives "public" and "private" methods based on whether the module uses the interface or the modport

SystemVerilog Assertions

What is an Assertion?

A concise description of [un]desired behavior



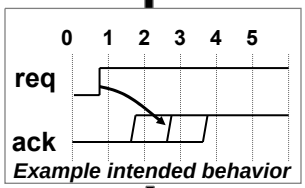
Example intended behavior

"After the request signal is asserted, the acknowledge signal must come 1 to 3 cycles later"

Concise and Expressive

SVA Assertion

```
property req_ack;
 @(posedge clk) req ##[1:3] $rose(ack);
endproperty
as_req_ack: assert property(req_ack);
```



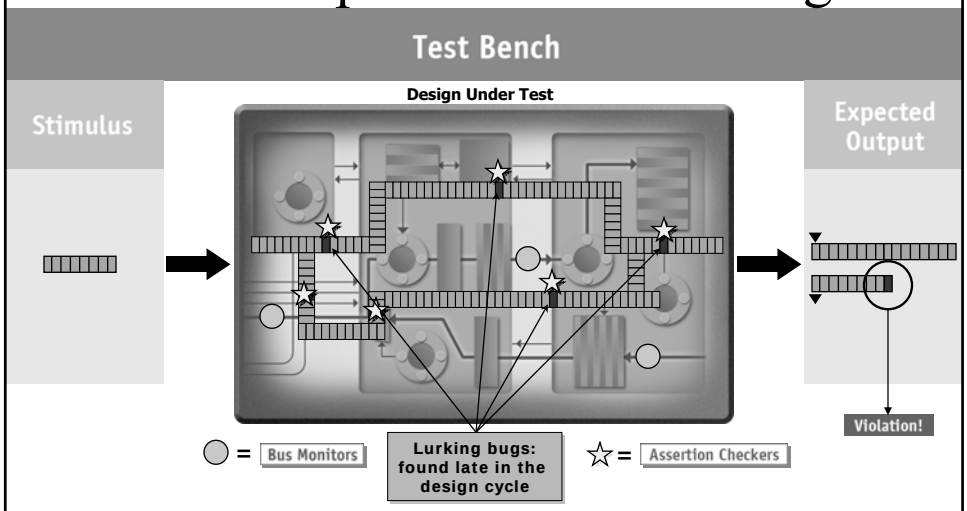
Example intended behavior

Verilog

```
always @(posedge req)
  begin
     repeat (1) @(posedge clk);
     fork: pos_pos
        begin
           @(posedge ack)
           $display("Assertion Success",$time);
            disable pos_pos;
        end
       begin
          repeat (2) @(posedge clk);
          $display("Assertion Failure",$time);
          disable pos_pos;
       end
     join
  end // always
```

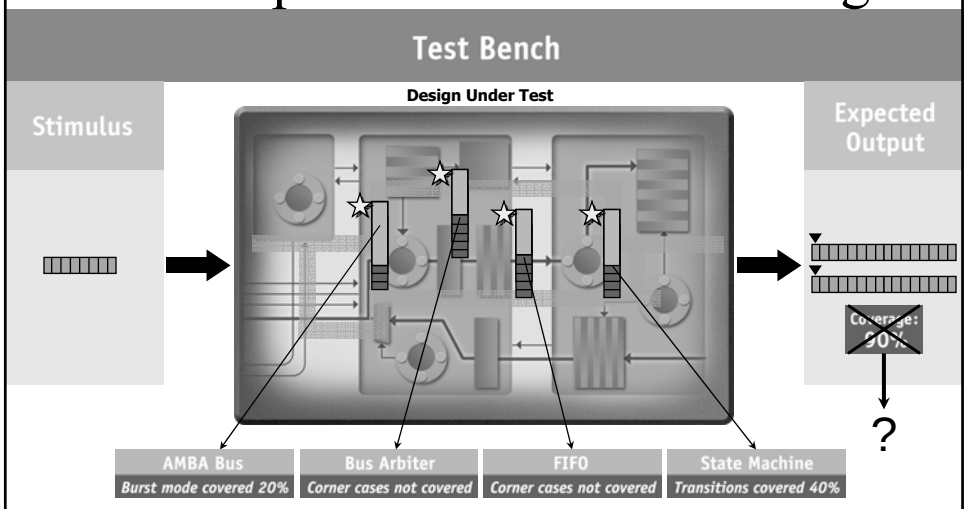
HDL Assertion

ABV Improves Time-To-Bug



- ABV detects bugs at the source, saving valuable debug time
- ABV detects bugs missed by top-level test benches

ABV Improves Time-To-Coverage



- ABV reveals internal structural coverage
- ABV produces actionable metrics to improve coverage

SV Assertions Flexible Use Model

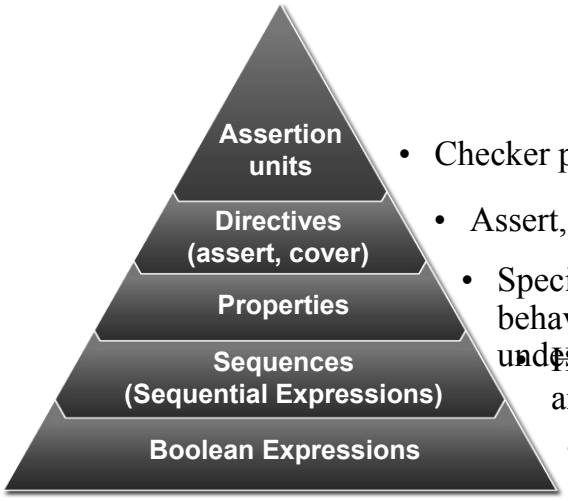
```
module arb (input req, gnt …);
property s1;
   (req && !gnt)[*0:5] ##1 gnt && req ##1 !req ;
endproperty
PA: assert property (s1);
endmodule
```

• Assertions can be embedded directly in verilog modules

```
program verify_arb;
property p1;
  @(posedge clk) ((reset == 1) && (mode == 1)
     && (st == REQ) && (!arb) && (foo)) |=> s1;
endproperty
DA: assert property (p1);
endprogram

Bind  arb arb_props arb1 (rdy, ack, …);
```

• Assertions can be coded in external files and bound to module and/or instance

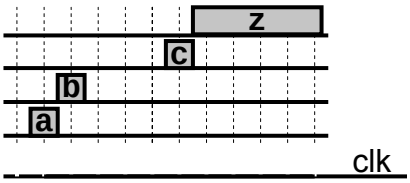
SV Assertion Language Structure



- Checker packaging
- Assert, assume, cover
- Specification of behavior; desired or undesired
- How boolean events are related over time
- True or False expression

Sequential Regular Expressions

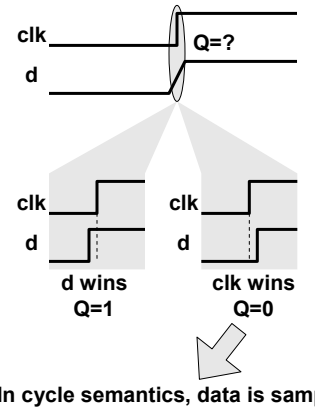
- Describing a sequence of events
  - Boolean expressions related over time
- Sequence Concatenation
  - Temporal delay
    - Concatenation character is ##n or ##[n:m]




```
sequence atest;
@(posedge clk)
    a ##1 b ##4 c ##[1:5] z;
endsequence
```


Assertions in Formal and Simulation

- Fundamental requirement for common semantics across tools
- Simulation is event-based
  - Subject to race conditions
- Formal (and Synthesis) are cycle-based
  - Clock always wins



In cycle semantics, data is sampled at the beginning of a timestep

This is equivalent to sampling the data at the end of the previous timestep

Sequences Encapsulate Behavior

- Can be Declared

```
sequence <name>[(<args>)];
  [@(<clocking>)] <sequence>;
endsequence
sequence s1(a,b);
  @(posedge clk) a[*2] ##3 b;
endsequence
```

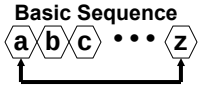
- Sequences Can Be Built From Other Sequences

```
sequence s2; @(posedge clk) c ##1 d;
endsequence
sequence s3; @(posedge clk) s1(e,f) ##1 s2;
endsequence
```

- Operations to compose sequences
  - and, or, intersect, within, throughout

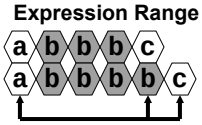
Sequence Examples

Basic Sequence



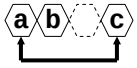

```
a ##1 b ##1 c …##1 z
```

Expression Range



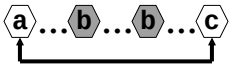

```
a ##1 b[*3:4] ##1 c
```

Sequence with Delay



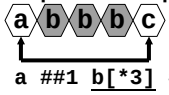

```
a ##1 b ##2 c
```

Expression Non-Consecutive "Counting" Repetition



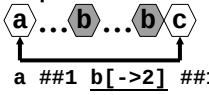

```
a ##1 b[=2] ##1 c
```

Expression Repetition




```
a ##1 b[*3] ##1 c
```

Expression "Goto" Repetition

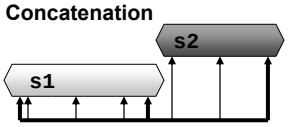



```
a ##1 b[->2] ##1 c
```


Multiple Clock Support

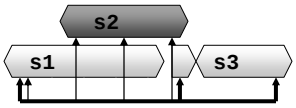
- Event controls must be specified for each subsequence

Multi-Clock Sequence Concatenation




```
@(clk1) s1 ##1 @(clk2) s2
```

Multi-Clock Sequence Matching




```
s1 ##1 s2.matched ##1 s3
```


Properties

```
property <name>[(<args>)];
[@(<clocking>)] <property_expr>;
endproperty
```

- Properties specify relationships between boolean expressions, sequences and subordinate properties over time (temporal relationships)
  - Statements of design, protocol, block or interface behavior over time
- Properties can be named and parameterized
- Properties can contain reset conditions (disable iff)
  - Evaluation of property is disabled
- Properties have their own operators
  - Implication (same or next cycle), Not, And, Or, Conditional (if else)
  - Recommend use of implication operators (`|->, |=>`) in properties

```
property rule1 (a, b, c, rst_n);
@(posedge clk) disable iff (!rst_n) a |-> b ##1 c;
endproperty

assert property rule1 (start, we, burst, reset_n);
```


Property implication

```
sequence_expr |-> [not] sequence_expr
sequence_expr |=> [not] sequence_expr
```

- |-> is overlapping implication
- |=> is non-overlapping implication
  same as:

```
sequence_expr ##1 `true|-> [not]
  sequence_expr
```

- Most commonly used to attach a precondition to sequence evaluation

SV Assertion Directives

- assert: Verify that a property holds (matches)

```
assert property (p1)action_block;
```

`action_block ::= [statement] [else statement]`

  - Action block
    - Executes in reactive region
    - Pass statement executes whenever property evaluates true else failure statement executes
      - $error is called if fail statement is absent
      - Recommend use $error instead of $display
- cover : Did a property hold (non-vacuously) at least once

```
cover property (p1)(statement_or_null);
```

  - Statement executes for each match of property
  - Can cover a sequence (Recommended)

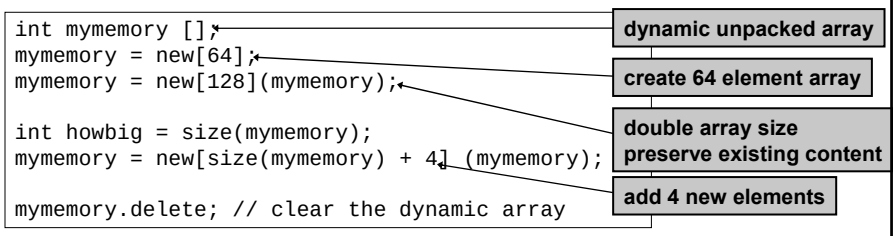
Strategies for Adopting ABV

- Specify design intent
  - What I thought I designed
    - Identify high-level elements in your blocks:
      - FIFOs, Arbiters, Memories, FSMs
    - Declarations
    - Key Operations
      - Put arithmetic overflow on arithmetic ops
      - Guard all module I/O
  - Corner cases
    - Make sure test exercises difficult scenarios
    - Make sure illegal situations are handled correctly
- Specify environment assumptions
  - What other blocks are doing
  - What the testbench *should* be doing
  - Avoid debugging false-negative testbench problems

SystemVerilog Verification Features

Arrays

- Dynamic sized arrays can be defined
  - Uses the `new[ ]` operator to size or re-size at run-time

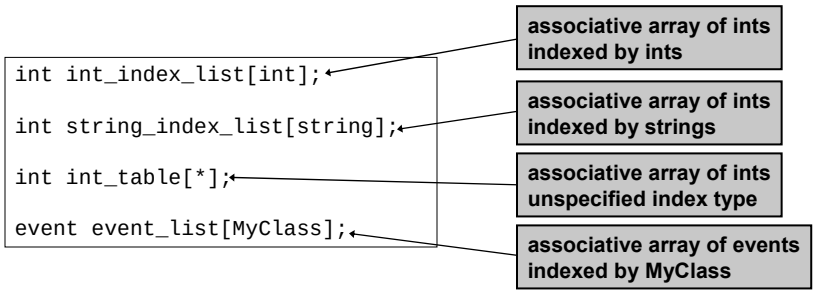


  - Dynamic arrays are always unpacked

Associative Arrays

- Indexed by content
  - Useful for modeling sparse memories, look-up tables, etc.

datatype `array_id[`***index_type***`];`



Associative Arrays

- Indexed by unspecified index (*)
  - The array can be indexed by any integral data type
  - 4-state index values containing X or Z are invalid
  - The ordering is unsigned numerical (smallest to largest)
- Indexed by string:
  - Indices can be strings or string literals of any length
  - An empty string “” index is valid
  - The ordering is lexicographical (lesser to greater)
- Indexed by class
  - Indices can be objects of that particular type or derived from that type
  - A null index is valid
  - The ordering is arbitrary but deterministic

Queues

- Analogous to variable-sized unpacked array
  - Grow and shrink automatically
  - Constant time access to the queue’s elements
  - Insertion and removal at the beginning and end supported
  - Elements are identified by an index
    - 0 represents the first
    - $ the last
- Can be manipulated like arrays

Queues

- Declaration
  - Default size is unbounded

```
int channel [$];
```

  - Can be initialized in declaration

```
byte datavalues [$] = '{2h'00, 2h'ff};
```

  - Can be bounded by specifying the right index

```
int q16int [$:15]; // queue limited to 16 integers
```

  - Can be manipulated via array/concatenation syntax

```
q = q[1:$]; // delete first entry
q = q[0:$-1]; // delete last entry
q = {q[0:n-2],q[n:$]}; //delete nth entry
q = {q[0:n-1],a,q[n:$]}; // insert a at q[n]
```

  - Built-in methods for push,pop,insert,delete,etc.

Directed vs. Constrained-Random

- Directed tests exercise a specific scenario
  - You direct the test
  - You explicitly orchestrate the interactions
  - It's a random world. What if you miss something?
- Injecting randomness exposes corner cases
- Multiple adoption strategies
  - Basic step: call directed tests in random order
    - Can't assume everything happens out of reset
  - Enhance directed tests to randomize more than just data
  - Build full CRV environment from scratch
- All strategies require functional coverage

☑ Let the Tool Find Cases You Haven't Thought Of

Terminology

- Pseudo-Random Data Generation
  - Using pseudo-random algorithms to generate "random" (but deterministic) numbers
- Directed Random Testing
  - A directed test in that uses pseudo-random numbers for parameters/data/etc.
  - Write random data to a random address
  - Read back from same address and compare
- Constrained-Random Testing
  - Random numbers are bound by arithmetic relationships to other variables
  - By randomizing high-level data types, the scenario exercised can be randomized as well
  - Randomly perform reads and writes with data and protocol mode dependent on address range

Random Stability

- The Random Number Generator (RNG) generates a pseudo-random sequence of

```
int A;
initial begin
  A =
$urandom('mySEED);
  repeat(5) begin
    A = $urandom;
    B = $urandom;
    $display(A);
    end
  end
```

```
int A;
initial
  repeat(5) begin
    A = $urandom;
    $display(A);
  end
initial
  repeat(5) begin
    B = $urandom;
    $display(B);
  end
```

Changing mySEED allows new RNG sequence for each run

Each run yields the same values: A = 6, 35, 41, 3, 27

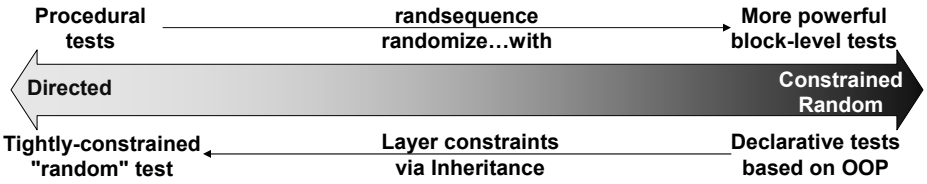
B "steals" values from the RNG stream: A = 6, 41, 27,…

Each process has its own statically-calculated seed

B stream is now independent of A and vice-versa

Randomization Strategies

- Depends on your existing strategy and your background
- Procedural randomization
  - Good for enhancing block-level directed tests
  - Doesn't require a lot of "software" knowledge
  - Limited reuse



- Declarative (object-oriented) randomization
  - "Knobs" are built into verification infrastructure
  - Requires investment to take advantage of powerful reuse capabilities

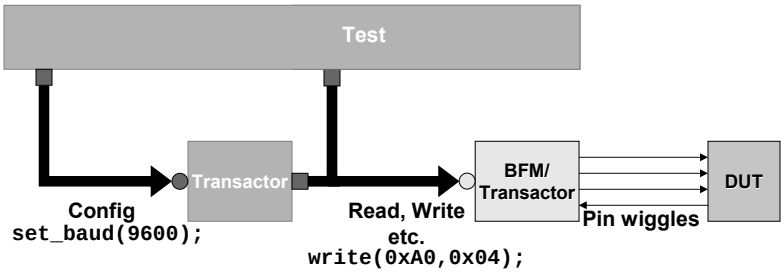
What to Randomize?

- Control
  - instructions
  - operations
  - device configuration
- Data
  - addresses
  - packets (src/dst, payload)
  - parameters
  - transaction types
- Time
  - clock cycles between events
  - how long to run test
- Error injection
  - Make sure the dut responds correctly

- Key is to create constraint sets that thoroughly explore relevant aspects of the design state-space
- Parameterized constraints allow scenarios to be tweaked on the fly

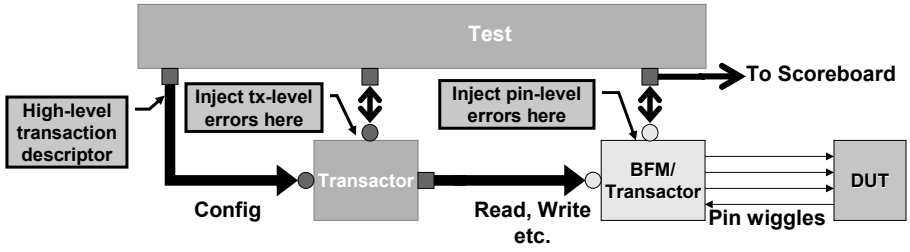
These aspects of a test can be randomized procedurally, or via Object-Oriented mechanisms

Transaction-Based Verification Interacting with the Design



- Test interacts with design via *transactors*
  - Transactors convert across abstraction layers
- TBV keeps test focused on *what* should happen
  - Isolates test from low-level implementation changes
  - Allows randomization of higher-level operations
  - Captures design intent in a more natural form
- Important concept whether using OO or not

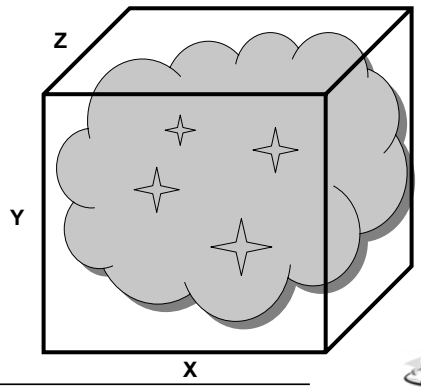
Error Injection



- Don't build all possible errors into the transaction descriptor
  - Makes it too big
  - Not reusable
- Inject errors appropriate to the downstream abstraction level of the transactor
- Control ports and callbacks let the test coordinate errors
  - Don't forget to let your scoreboard know when errors are expected

What is a Random Constraint Solver?

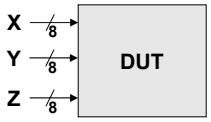
- Declare a set of random variables – X, Y, Z
- Declare a set of constraints – $Y < 42$, $X \leq Y \leq Z$
- Find the set of values that meet the given constraints
- Randomly pick solutions



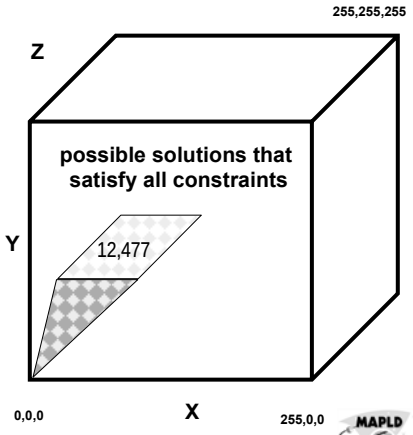
Computing the Solution Space

Solutions

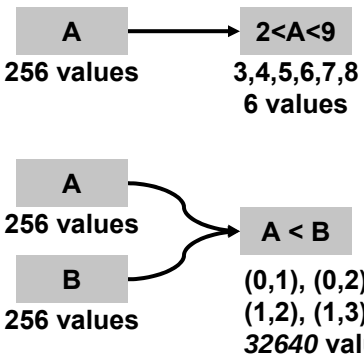
```
# 21 26 32
# 1d 22 91
# 04 0e d0
# 0f 28 60
# 14 20 d1
# 07 1f 80
```



Constraint	Number of Solutions
Unconstrained	$2^{8+8+8} = 2^{24} = 16{,}777{,}216$
$Y < 42$	$42*2^{8+8} = 2{,}752{,}512$
$X \leq Y \leq Z$	$\frac{1}{3}*\frac{1}{2}*2^{24} = 2{,}796{,}202$
X[7:4]=Z[3:0]	$2^4*2^{8+4+4} = 2^{20}$ 1,048,576



Random Variables and the Solution Space



8 bit random variable

Small set of constraint relationships

16 bit random variable

Large set of constraint relationships

- Need to be aware of the complexity of constraints

Solution Space

- Implication operator `->`

```
rand bit s;
rand bit [2:0] d;
constraint cons { s -> d == 0;}
```

- Constraint reads if "s" is true then d is 0
  - as if "s" determines "d"
  - Actually, `->` is bidirectional, s and d are determined together
- Possible 9 values in the solution space are :

```
if s = 0       d = 7 or 6 or 5 or 4 or 3 or 2 or 1 or 0
if s = 1       d = 0
```

  - The (s,d) pairs will be (0,0), (0,1), (0,2),(0,3),(0,4),(0,5), (0,6),(0,7) and (1,0)
  - Probability of picking s = 1 is 1 in 9
- To keep the solution space the same and pick "s" true with a probability of 50%

```
constraint cons_plus {solve s before d;}
```

Random Weighted Case

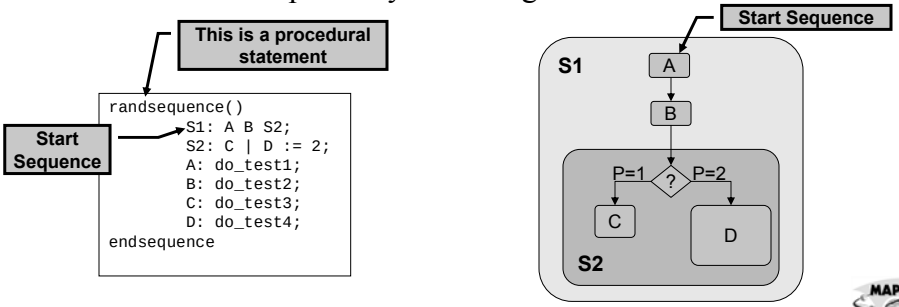
- SystemVerilog allows for a case statement that randomly selects one of its branches
- Each case can have a specific weight which can be numeric or an expression

```
int a,b,c;

initial begin
  a = 3;
  b = 2;
  randcase
    1   : c = 12;     //              10% probability
    a   : c = 20;     // weight is 3, 30% probability
    a-b : c = 99;     // weight is 1, 10% probability
    5   : c = 101;    //              50% probability
  endcase
end
```

  - Number of choices is static
  - Relative probability of each choice can be dynamic

Sequences

- Implemented in a randsequence() block
- Functionality of the sequence block described as a grammar
  - Composed of rules and productions
  - Productions are classified as terminal or non-terminal
  - Stream is composed by streaming items



Generating Test Scenarios

- Random Sequences specify a set of test scenarios that make sense
  - Multiple control mechanisms for production statements
    - if..else, case, repeat, return, exit
  - Control mechanisms and weights use expressions
    - Can change dynamically during a test
    - Allow test to react to DUT responses or other criteria
- Each "walk" through creates a different test scenario
- Each scenario can be as atomic as you want it
  - Explicit directed test
  - Execute CPU instruction (generate meaningful code structures)
  - Send packet of explicit type (send valid stream of random packets)
- Since it's a procedural statement, randsequence is easily added to existing directed tests

Specifying Constraints

- SystemVerilog adds in-line constraint

```
initial begin
  for(i=0;i<32;i++) begin
    addr = i;
    data = $random();
    if(addr < 16)
      data[7] = 1'b1;
    do_write(addr,data);
  end
  for(i=0;i<32;i++) begin
    addr = i;
    do_read(addr,data);
    assert(data == exp[i]);
  end
end
```

```
initial begin
  for(i=0;i<32;i++) begin
    randomize(addr,data) with {addr < 96;
            if(addr<16) data[7] == 1'b1;};
    addr_list = '{addr_list,addr};
    do_write(addr,data);
  end
  for(i=0;i<32;i++) begin
    addr = pick_addr(addr_list);
    do_read(addr,data);
    assert(data == exp[addr]);
  end
end
```


SystemVerilog Testbench Automation

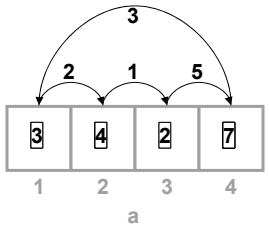
Functional Coverage Analysis

- Coverage analysis depends on what you're looking for
  - Have I exercised all transaction types on the bus?
    - Control-Oriented (in bus monitor)
  - Have I generated packets of every type/length
    - Data-Oriented (in testbench)
  - Did I successfully model all transaction types to every interesting address range?
    - Combination of control- and data-oriented coverage
- Must be able to assemble a verification infrastructure to gather the information to answer the questions

SystemVerilog Provides Coverage Tools

- Data-oriented functional coverage
  - **`covergroup`**
  - Records data values at interesting times
- Control-oriented functional coverage
  - **`cover`** sequences/properties
  - Concise description of temporal behaviors
- Data- and Control-oriented functional coverage work together
  - Properties detect the temporal activity
  - Covergroup captures transaction attributes
  - SystemVerilog supports communication between them

Data-Oriented Coverage Recording

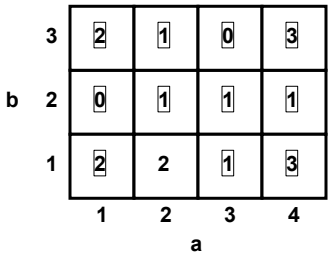


Transition Coverage: How many times did a==2 after a==1?

Simple Coverage: How many times did a==1?

- Records the values of variables at a particular simulation time
- Usually captured by the testbench
- Many possibilities for analysis

Data-Oriented Coverage Recording



Cross Coverage: How many times did a==2 while b==1?

- Records the values of variables at a particular simulation time
- Usually captured by the testbench
- Many possibilities for analysis
- The key is to know when to sample the data

Covergroup

```
bit [7:0] addr;
enum {WRITE, READ} kind;
int dly;
covergroup mycov @smp_event;
   coverpoint addr {bins a[4] = {[0:31]};}
   coverpoint kind {bins k[] = {WRITE,READ};}
   coverpoint dly {bins d[] = {[1:4]};}
   addr_kind: cross addr, kind;
   kind_dly: cross kind, dly;
endgroup
mycov covgrp = new;
```

4 address ranges of interest

2 kinds of transaction

Delay range

Transaction type vs. addr range (8 bins)

Transaction type vs. delay (8 bins)

- Automatically samples data values at specified time/event
- Specifies “bins” to count number of samples in a particular value range
- Gives information about the recorded data values
  - You have to know what questions you want to answer
  - Build your covergroup to answer the questions

Type vs. Instance Coverage

- Type coverage is cumulative for all instances of a covergroup
  - c1::a.get_coverage();

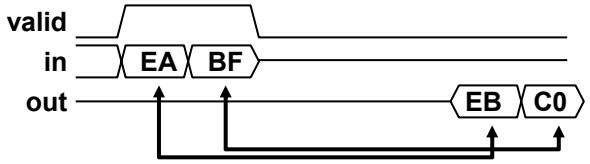
```
covergroup c1 (ref int a) @(posedge clk);
      coverpoint a;
endgroup
initial begin
   c1 group1 = new(instance1.int_var);
   c1 group2 = new(instance2.int_var);
end
```

- Instance coverage is for a particular instance
  - group2.a.get_inst_coverage()
- Global cumulative coverage for all covergroup instances
  - $get_coverage();

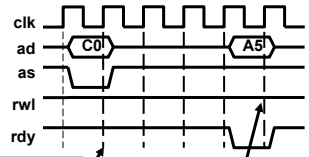
Assertion Hooks for Coverage

- SystemVerilog lets you capture data at critical points within a sequence/property
  - Local variables

```
property e;
  int x;
  (valid, x = in) |-> ##5 (out ==
 (x+1));
endproperty
```



Local Variables
Capture Transient Data




```
property trans;
   a_t paddr;
   tr_t pkind;
   int pdly;
   @(posedge clk) ((as == 1'b0), pkind = tr_t'(rwl),paddr = ad,pdly=0)
                  |=> ((as == 1'b1 && rdy == 1'b1, ++pdly)[*1:4])
                  ##1 ((as == 1'b1 && rdy == 1'b0), docov(paddr,pkind,pdly));
endproperty

cover property (trans);

function void docov(input a_t aval, tr_t kval, int dlyval);
   addr = aval;
   kind = kval;
   dly = dlyval;
   covgrp.sample; // immediate sample in covergroup
endfunction
```

Capture addr, kind here

Call coverage here

init delay counter

incr delay counter

Only called when property is successful

Function passes local variables outside property

Local Variables
Capture Transient Data

```
covergroup mycov @smp_event;
   coverpoint addr {bins a[4] = {[0:31]};}
   coverpoint kind {bins k[] = {WRITE,READ};}
   coverpoint dly {bins d[] = {[1:4]};}
   addr_kind: cross addr, kind;
   kind_dly: cross kind, dly;
endgroup
mycov covgrp = new;
```

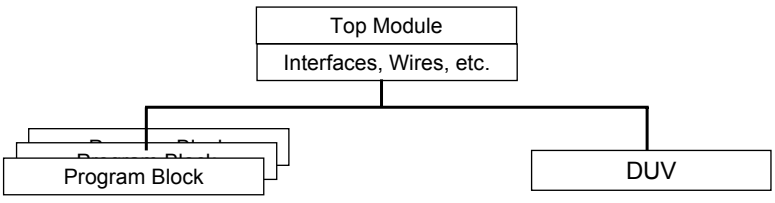
```
property trans;
   a_t paddr;
   tr_t pkind;
   int pdly;
   @(posedge clk) ((as == 1'b0), pkind = tr_t'(rwl),paddr = ad,pdly=0)
                  |=> ((as == 1'b1 && rdy == 1'b1, ++pdly)[*1:4])
                  ##1 ((as == 1'b1 && rdy == 1'b0), docov(paddr,pkind,pdly));
endproperty

cover property (trans);

function void docov(input a_t aval, tr_t kval, int dlyval);
   addr = aval;
   kind = kval;
   dly = dlyval;
   covgrp.sample; // immediate sample in covergroup
endfunction
```

Program Blocks

- Used to encapsulate testbench functionality
- Always a leaf in the hierarchy
- Decouples the verification code from DUT
  - DUT cannot reference anything in program block
  - Program blocks can scope into each other and DUT
- Avoids DUT dependency on TB
- Helps prevent DUT/TB race conditions
- No limit on number of program blocks

Program Blocks

- Allows for code reuse
  - Signal references are local to the ports on the program
  - Gives multiple threads of execution
  - initial and final blocks can be defined
  - Forms heart of the test flow

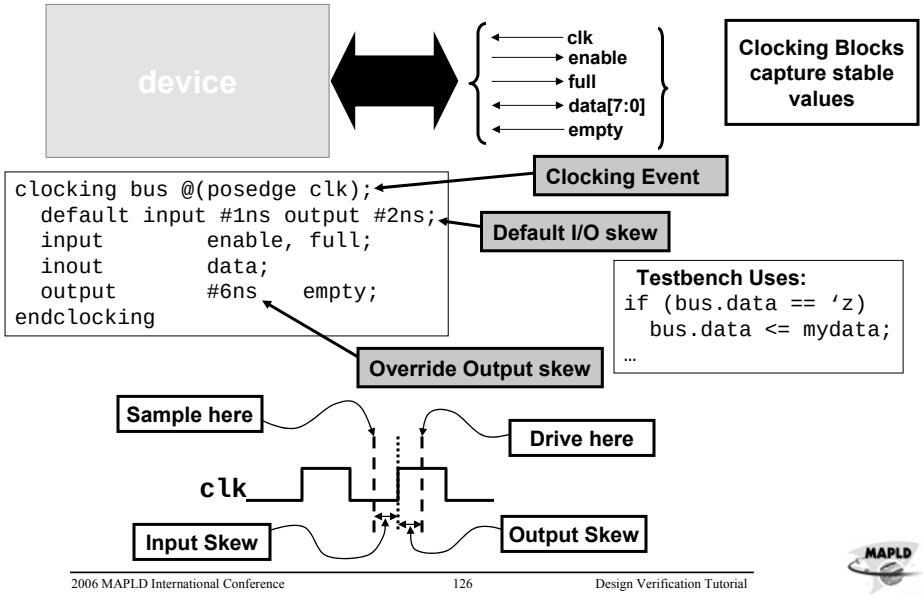


Program Blocks

- Simple example

```
program cpu_test(interface cpu);
class intel_mgmt;
          // AC Timing Characteristics ...
          static int t1 = 10;
          task reset;
                    cpu.Sel  = 1 ;
                    cpu.Rd   = 1 ;
                    cpu.Wr   = 1 ;
                    cpu.Data_drive = 16'hzzzz ;
                    cpu.rst  = 0;
                    # t1;
                    cpu.rst  = 1;
                    # t1;
          endtask
endclass
intel_mgmt the_CPU = new();
initial
begin
          $write("Starting CPU test\n");
          the_CPU.reset;
          the_CPU.write(1, 10);
          //etcetera
end
endprogram
```


Clocking Blocks



Object-Oriented Programming

- Organize programs in the same way that objects are organized in the real world
- Break program into blocks that work together to accomplish a task, each block has a well defined interface
- Focuses on the data and what you are trying to do with it rather than on procedural algorithms

- Class – A blueprint for a house
  - Program element "containing" related group of features and functionality.
  - Encapsulates functionality
  - Provides a template for building objects
- Object – The actual house
  - An object is an instance of a class
- Properties – It has light switches
  - Variables specific to the class
- Methods – Turn on/off the lights
  - Tasks/functions specific to the class

Class Basics

- Class Definitions Contain Data and Methods
- Classes Are Instantiated Dynamically to Create *Objects*
  - *Static* members create a single element shared by all objects of a particular class type
- *Objects* Are Accessed Via *Handles*
  - Safe Pointers, Like Java
- Classes Can *Inherit* Properties and Methods From Other Classes
- Classes Can Be Parameterized

The SystemVerilog Class

Basic class syntax:

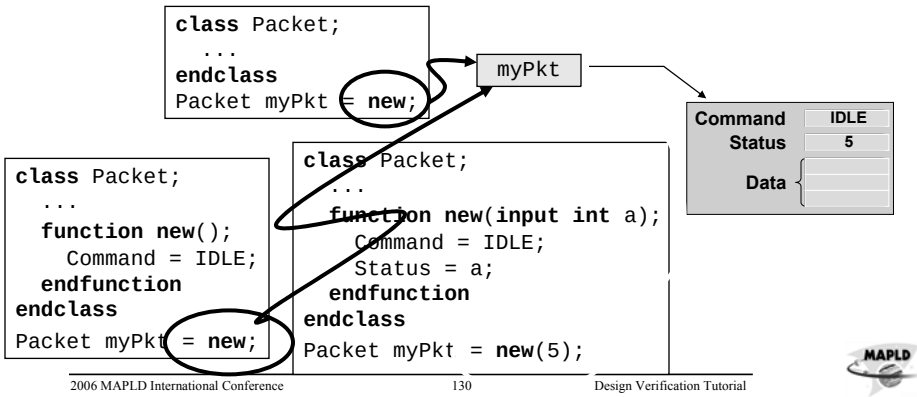
```
class name;
    <data_declarations>;
    <task/function_declarations>;
endclass
```

Note:
Class declaration does not allocate any storage

```
class Packet;
  cmd_t Command;
  int Status;
  struct Data;
  function int GetStatus();
    return(Status);
  endfunction
  task SetCommand(input cmd_t a);
    Command = a;
  endtask
endclass
```


Instantiating SV Classes

- Classes are dynamically created objects
  - Every object has a built-in "new()" constructor method
  - Can also create user defined constructor that overloads built-in



Handling Memory with SV Classes

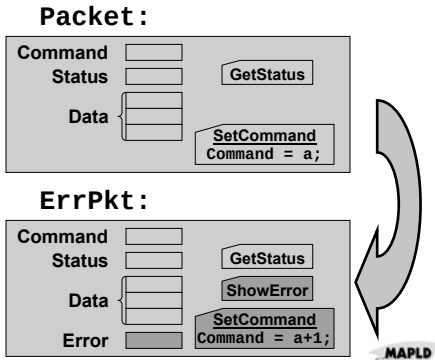
- Object Destruction/De-allocation done automatically when an object is no longer being referenced
  - NO destructors
  - NO memory leaks
  - NO unexpected side effects
- Automatic garbage collection

```
Packet Pkt1 = new();
Packet Pkt2 = new();
initial begin
     ...
  Send_Pkt( Pkt1 );
     ...
  Send_Pkt( Pkt2 );
     ...
  Pkt1 = null; Pkt2 = null;
end
```

Extending SV Classes

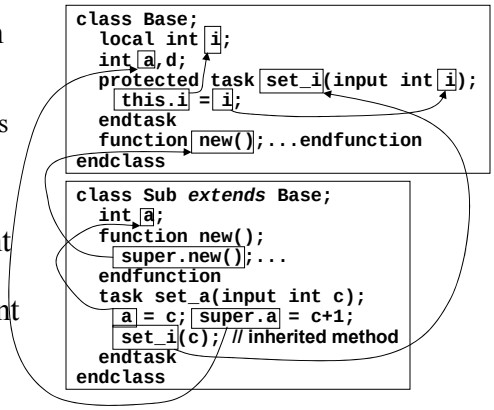
- Classes inherit properties & methods from other classes
  - Subclasses can redefine the parent's methods explicitly
  - Allows customization without breaking or rewriting known-good functionality in the parent class

```
class ErrPkt extends Packet;
  bit Error;
  function bit ShowError();
    return(Error);
  endfunction
  task SetCommand (input cmd_t a);
    Command = a + 1;
  endtask
endclass
```



Class Hierarchy

- Class members can be hidden from external access
  - **`local`** members can only be referenced from within the class
  - **`protected`** members can be referenced from within a subclass
- **`this`** pointer refers to current instance
- **`super`** pointer refers to parent class



```
class Base;
  local int i;
  int a,d;
  protected task set_i(input int i);
    this.i = i;
  endtask
  function new();...endfunction
endclass

class Sub extends Base;
  int a;
  function new();
    super.new();...
  endfunction
  task set_a(input int c);
    a = c; super.a = c+1;
    set_i(c); // inherited method
  endtask
endclass
```


Class Hierarchy

- Class members can be hidden from external access
  - **`local`** members can only be referenced from within the class
  - **`protected`** members can be referenced from within a subclass
- **`this`** pointer refers to current instance
- **`super`** pointer refers to parent class

```
class Base;
  local int i;
  int a,d;
  protected task set_i(input int i);
    this.i = i;
  endtask
  function new();...endfunction
endclass

class Sub extends Base;
  int a;
  function new();
    super.new();...
  endfunction
  task set_a(input int c);
    a = c; super.a = c+1;
    set_i(c);// inherited
  endtask
endclass
```

```
Sub S = new;
initial begin
  S.i = 4; // illegal – i is local to Base
  S.set_i(4);// illegal – set_i is protected
  S.a = 5; // legal – Base::a is hidden by Sub::a
  S.set_a(5);// legal – set_a is unprotected
  S.d = 3;// legal – d is inherited from Base
end
```


Parameterized Classes

- Allows Generic Class to be Instantiated as Objects of Different Types
  - Uses module-like parameter passing

```
class vector #(parameter int size = 1;);
  bit [size-1:0] a;
endclass
vector #(10) vten; // object with vector of size 10
vector #(.size(2)) vtwo; // object with vector of size 2
typedef vector#(4) Vfour; // Class with vector of size 4
```

```
class stack #(parameter type T = int;);
  local T items[];
  task push( T a ); ... endtask
  task pop( ref T a ); ... endtask
endclass
stack i_s; // default: a stack of int's
stack#(bit[1:10]) bs; // a stack of 10-bit vector
stack#(real) rs; // a stack of real numbers
stack#(Vfour) vs; // stack of classes
```

Avoid Writing Similar Code More than Once

Constraints and OOP

- Constraints are part of the OO data model

```
class TBase;
  rand logic [3:0] a;
  rand logic [3:0] b;

  constraint c1 { a < 4'b1100; }
  constraint c2 { b < 4'b1101; }
endclass
```

Declare random variables

Declare constraints

- Layer with inheritance

```
class TDerived extends TBase;
  constraint c3 { a > b ; }
  constraint c4 { a < b ; }
  constraint c5 {  a + b == 4'b1111; }
endclass
```

Add additional constraints

Note that c3 and c4 are mutually exclusive

- Change constraints on the fly with constraint_mode() and rand_mode()

```
TDerived derived = new();
derived.c3.constraint_mode(0); // Turn c3 off
status = randomize(derived);   // c1,c2,c4,c5 active
derived.b.rand_mode(0);        // Turn b's randomization off
```


In-line Random Variable Control

- When ***randomize()*** is called without arguments, only random variables are assigned values
- By providing arguments, only values within the argument list are randomized

```
class packet {
  rand byte src;
  rand byte dest;
  byte payload;  //payload is not a random byte
endclass

packet p1 = new;

initial begin
  p1.randomize();         // randomizes src and dest only
  p1.randomize(payload);  // randomizes payload only
  p1.randomize(src,payload);      // randomizes src and payload
  p1.randomize(null);     // returns function success only
end
```


State-Dependent Constraints

- Constraints based on other values allow expressions to change dynamically

```
class myState;
  bit [7:0] st = 0;
  rand enum kind {READ, WRITE};
  constraint rdfirst {if (st < 10) kind == READ;
                      else kind == WRITE;}
endclass
```

- Constraints can specify FSM-like behavior for random variables

```
class myState
  typedef enum StType {INIT, REQ, RD...};
  rand StType state = INIT;
  StType pstate;
  bit req;
  constraint fsm { if(pstate == INIT) {state == REQ; req == 1;};
                   if(pstate == REQ && rdwr == 1) state == Rd;
                   ...};
endclass
```


Built-in Methods

- **`randomize()`** automatically calls two built-in methods
  - **`pre_randomize()`:** Called before randomization
  - **`post_randomize()`:** Called after randomization
- These methods can be used as "hooks" for the user to tap to peform operations such as:
  - Setting initial values of variables
  - Performing functions after the generator has assigned random values

`pre_randomize()` Method

- Users can override pre_randomize() in a class to set pre-conditions before the object is randomized

```
class MyPacket extends Packet {

  int packet_size;
  rand byte payload [];

  function void pre_randomize();
    super.pre_randomize();
    packet_size = 10;          //initialize packet_size
    payload = new[packet_size];
    …
  endfunction
endclass
```

Call to parent's **`super.pre_randomize()`** must be present, otherwise their pre-randomization processing steps shall be skipped

`post_randomize()` Method

- Users can override **`post_randomize()`** in a class to perform calculations on generated data, print diagnostics, and check post-conditions

```
class MyPacket extends Packet {
  byte parity;

  function void post_randomize();
    super.post_randomize();
    parity = calculate_parity();  //calculate parity …
  endfunction

  function byte calculate_parity();
       …
  endfunction
endclass
```

Call to parent's **`super.post_randomize()`** must be present, otherwise their post-randomization processing steps shall be skipped

Virtual Interfaces

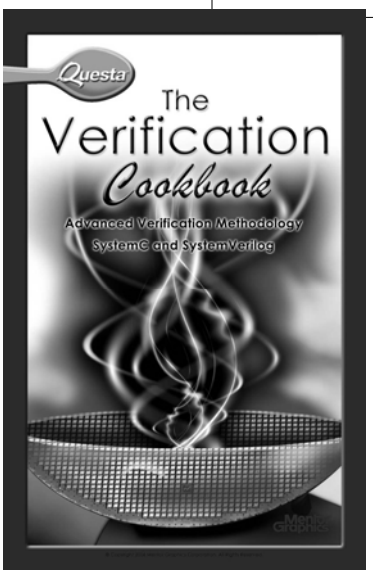
- Need a way to connect classes to actual interface signals
- Virtual Interface serves as a pointer to physical interface

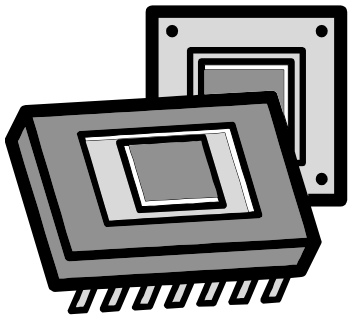
```
class BusDriver;
  virtual myBus bus;
  task send2bus(...);
    @(posedge bus.clk)
      bus.req <= 1'b1;
  ...
  endtask
endclass
```

```
program prog(myBus bif);
  BusDriver myDriver = new();
  initial begin
    myDriver.bus = bif;
    …
  end
endprogram
```

Putting it All Together: The Advanced Verification Methodology

Baking a *Chip*





Introducing the Advanced Verification Methodology

- The Open-Source AVM Library – The Ingredients
  - Library of modular, reusable verification components
    - Implemented in both SystemVerilog and SystemC
    - Consistent Transaction-Level interfaces with common semantics
  - Infrastructure details built-in so you don't have to worry about them
    - Simple connections between components
    - Controllable, customizable error/message reporting
  - Open-Source means you are free to use them, with full access to all the source code
- The Verification Cookbook – The Recipes
  - Open-source runnable examples
    - Illustrate concepts and serve as templates for your use
  - Examples build on previous ones to introduce advanced verification concepts incrementally

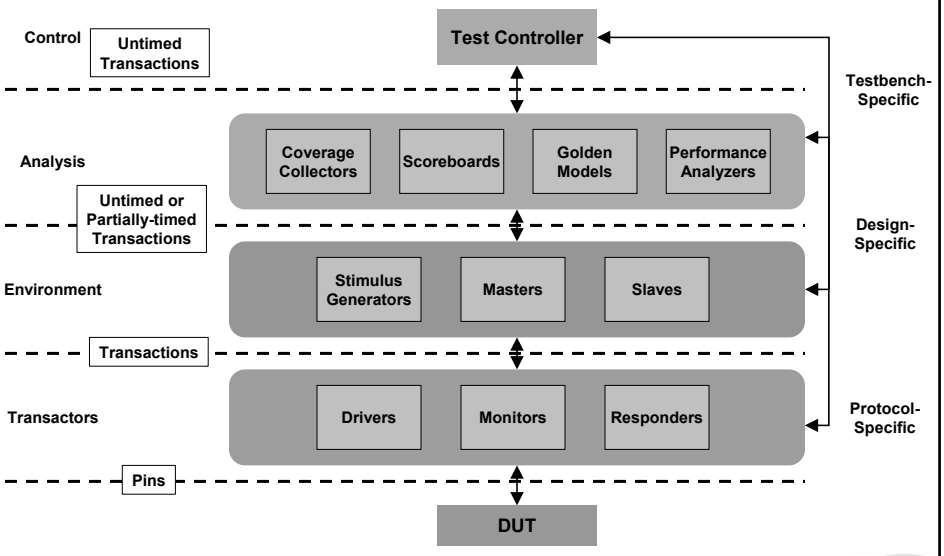
Verification Methodology

- Understand the process
  - Compare observed DUT behavior against expected behavior
  - Many ways to specify expected behavior
- Get the ingredients
  - AVM stocks your shelves
  - Basic staples (transactions, stimulus generator, etc.)
  - Application-specific stuff (constraints, coverage, etc.)
- Follow the recipe
  - Assemble the verification components into an environment
  - Generate proper stimulus and automatically check results
  - Gather coverage and track progress
- Presentation
  - Analyze results to identify areas of weakness
  - Tweak constraints to get better coverage

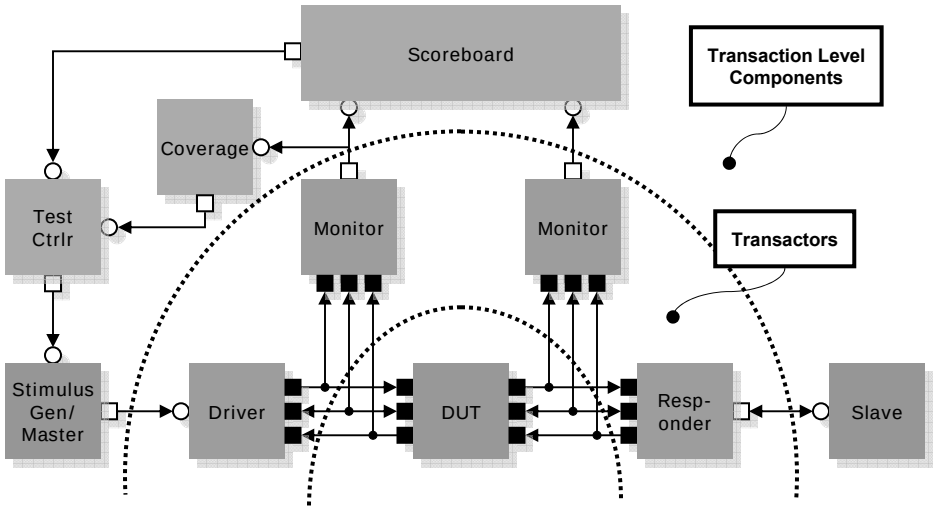
Key Aspects of a Good Methodology

- Automation
  - Let the tools do the work
- Observability
  - Self-checking is critical
  - Automate self-checking and coverage tracking
  - Assertion-based Verification
- Controllability
  - Make sure you exercise critical functionality
  - Constrained-Random stimulus and formal verification automate the control process
- Reusability
  - Don't reinvent the wheel
  - Reusability is functionality- and/or protocol-specific
  - Reuse is critical across projects, across teams, and across tools
- Measurability
  - If you can't measure it, you can't improve it
  - Tools automate the collection of information
  - Analysis requires tools to provide information in useful ways
  - All tools must consistently contribute to measurement and analysis

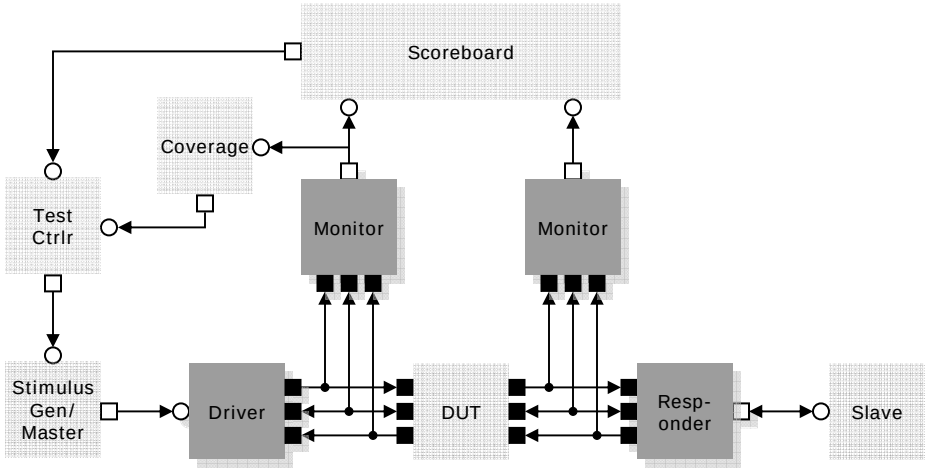
AVM Layering



AVM Architecture

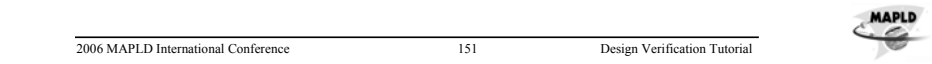


Transactors

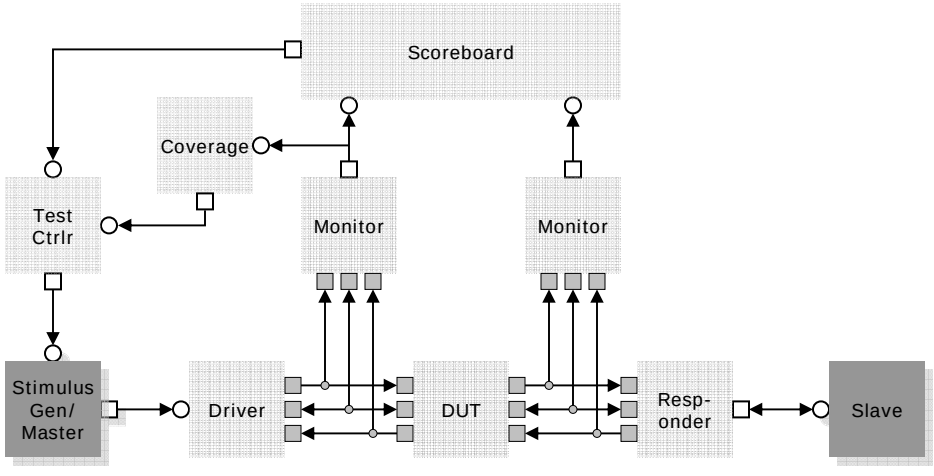


Transactors

- A Transactor
  - Converts traffic between signal and transaction domains
- A Monitor
  - Converts signal level traffic to a stream of transactions
    - Checks for protocol violations
    - Moves traffic from the design to analysis portion of the TB
- A Driver
  - Is a Transactor that takes an active part in the protocol
    - Interprets transactions and drives signal level bus
    - May be bi-directional
- A Responder
  - Is the mirror image of a Driver. It plays an active part in the protocol.
    - It identifies a response and forwards it to the slave
    - It takes the response from the slave and applies it to the bus



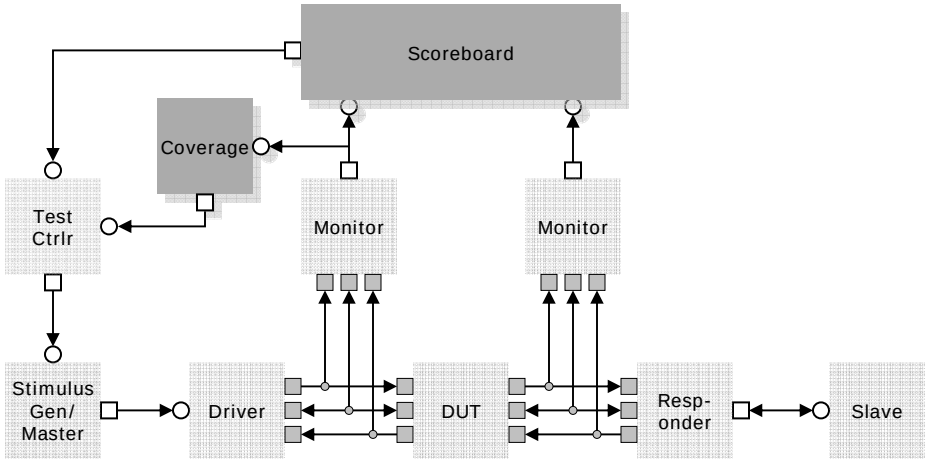
Environment



Environment

- Stimulus Generator
  - Generates sequences of transactions that are sent into the testbench
  - Generate constrained random stimulus; or
  - Generate directed stimulus
- Master
  - Bi-directional component
  - Initiates activity
- Slave
  - Bi-directional component
  - Response to activity

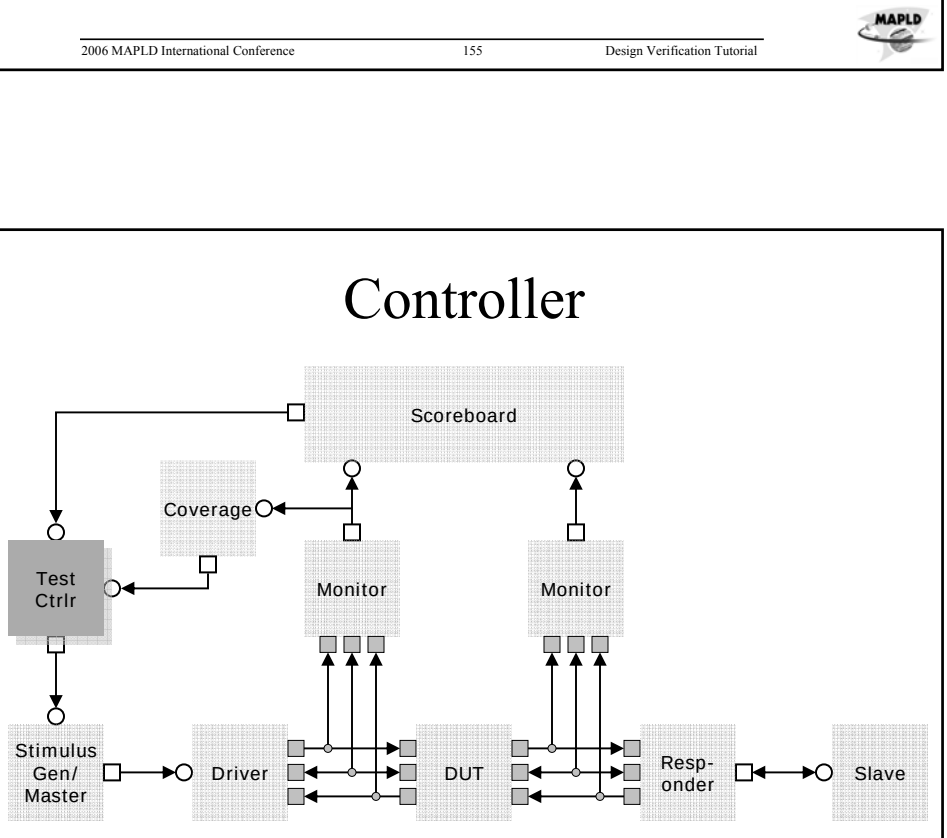
Analysis Components



Analysis

- Consume transactions from monitors
- Perform data reduction of some sort
- Operate at the transaction level

- Scoreboard
  - Checks the intended functionality of the DUT
- Coverage Collector
  - Counts things



Control

- Supplies configuration information to verification components
- Schedules the different stimulus generators
- Monitors the functional coverage
  - Are we done?
- Tests the state of the scoreboards
  - To detect functional errors
- Design-specific component

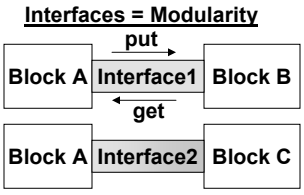
Design Intent

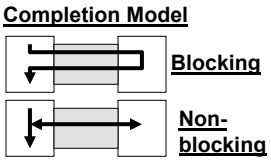
- Design representation is well understood…
- How to represent intent?

- Expression of intent must:
  - be easier/faster to build than the design
  - be at a higher abstraction level than the design
  - communicate with the design

TLM in a Nutshell

- *Initiator* puts/gets transaction to/from a *target*
  - Initiator port *requires* an interface
  - Target export *provides* the implementation of the interface
- Completion Model can be *blocking* or *nonblocking*
  - Blocking methods may be tasks
  - Nonblocking must be functions





Direction

- Unidirectional Dataflow
  - Choose put, get or peek to move data between Verification Components

```
p.put( req );
```

```
p.get( rsp );
```

```
p.peek( rsp );
```

- Bidirectional Dataflow layered on top of unidirectional
  - put + get for pipelined buses
  - transport for non pipelined

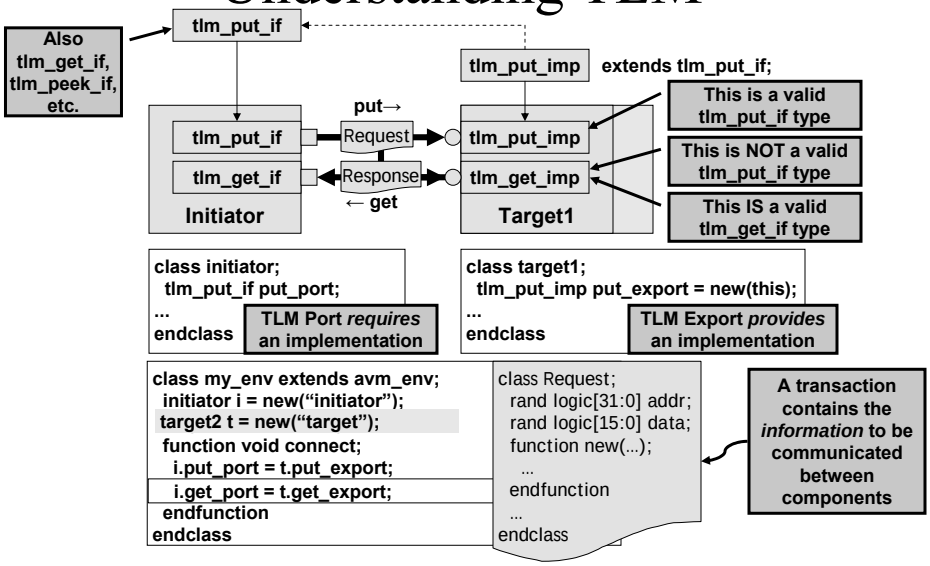
TLM also has equivalent non blocking APIs for speed

```
p.put( req );
p.get( rsp );
```

OR

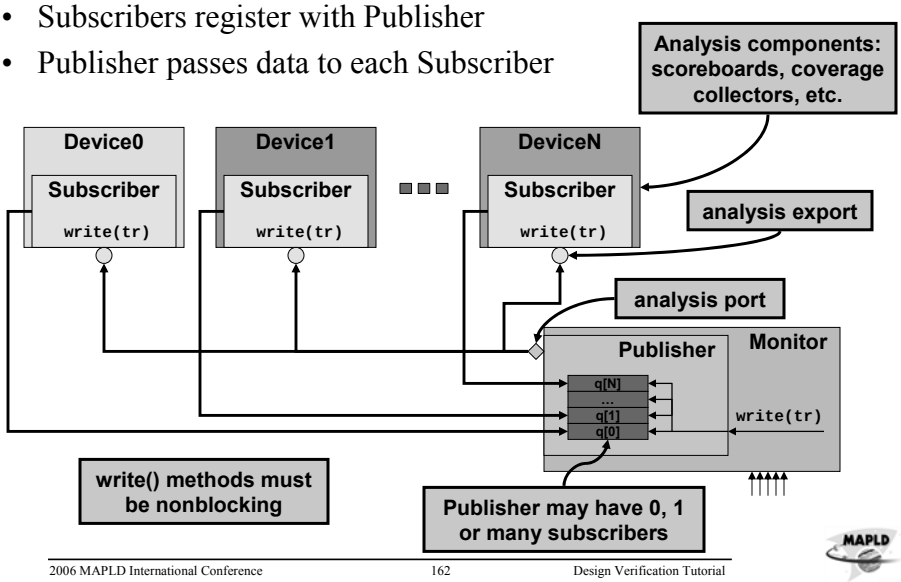
```
p.transport( req , rsp );
```

Understanding TLM

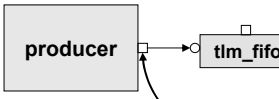


Analysis Port Description

- Subscribers register with Publisher
- Publisher passes data to each Subscriber



Simple Fifo Example




```
class producer extends verif_component;
  tlm_put_if #( int ) put_port;

  task run();
    for( int i = 0; i < 10; i++ )
      begin
      $display("about to put %d",i);
      put_port.put( i );
    end
  endtask;

endclass
```

```
class consumer extends verif_component;
  tlm_get_if #( int ) get_port;

   task run();
    int i;
    forever begin
      get_port.get( i );
      $display( "Just got %d" , i );
    end
  endtask

endclass
```


Simple Fifo Example

```
program top;
  env = new();

  initial begin
    env.connect();
    verif_component::run_all_threads();
  end
endprogram
```

Elaboration

Execution

```
class env;
  producer p = new;
  consumer c = new;
  tlm_fifo #( int ) f = new;

  function void connect;
    p.put_port = f.blocking_put_export;
    c.get_port = f.blocking_get_export;
  endfunction
endclass
```

Binding

```
class producer extends verif_component;
  tlm_put_if #( int ) put_port;

  task run();
    for( int i = 0; i < 10; i++ )
      begin
      $display("about to put %d",i);
      put_port.put( i );
    end
  endtask;

endclass
```

```
class consumer extends verif_component;
  tlm_get_if #( int ) get_port;

   task run();
    int i;
    forever begin
      get_port.get( i );
      $display( "Just got %d" , i );
    end
  endtask

endclass
```

TLM Channels

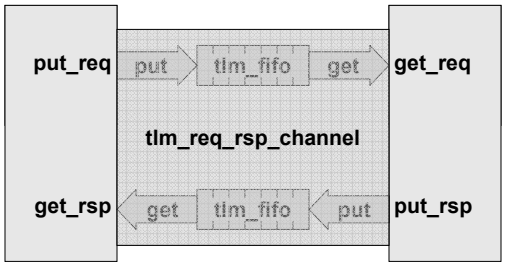
- tlm_fifo
  - unidirectional transactions between free-running independent components
- tlm_req_rsp_channel
  - two back-to-back tlm_fifos
  - pipelined or out-of-order protocols
- tlm_transport_channel
  - tlm_req_rsp_channel with fifo size = 1
  - non-pipelined and in-order

```
class tlm_fifo #(type T,
                 int BOUND=1);
  local mailbox #(T) m;
  extern task put(input T t);
  extern task get(output T t);
  extern task peek(output T t);
  ...
endclass
```

```
class tlm_req_rsp_channel
    #(type REQ, RSP, int BOUND=1);
  tlm_fifo #(REQ, BOUND) req = new();
  tlm_fifo #(RSP, BOUND) rsp = new();
  ...
endclass
```

```
class tlm_transport_channel
    #(type REQ, RSP);
  tlm_fifo #(REQ, 1) req = new();
  tlm_fifo #(RSP, 1) rsp = new();
  ...
  task transport(input REQ reqt,
                 output RSP rspt);
    req.put(reqt);
    rsp.get(rspt);
  ...
endclass
```


TLM Channels



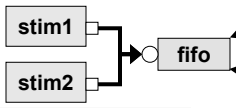
Exports in tlm_fifo

- May have multiple ports connected to them

```
program test;
  stimulus #(trans) stim1 = new();
  stimulus #(trans) stim2 = new();
  tlm_fifo #(trans) fifo = new();

  initial begin
    stim1.blocking_put_port = fifo.blocking_put_export;
    stim2.blocking_put_port = fifo.blocking_put_export;
    ...
  end
endprogram
```

```
class tlm_fifo #( T = int )
  tlm_blocking_put_imp #( this_type , T )
            blocking_put_export = new( this );

  task put( input T t ); … endtask
endclass
```

```
class stimulus #(T = int);
  tlm_blocking_put_if #(T) blocking_put_port;
  task send( input T t );
    blocking_put_port.put(t);
  endtask
endclass
```



Multiple puts will get resolved and interleaved automatically by the fifo

Fairness of interleaving is determined by the underlying mailbox

User may also control interleaving by turning generators on and off via the test controller

Analysis Fifo in SystemVerilog

- Specialization of tlm_fifo
  - Infinite fifo to ensure nonblocking writes
  - Still has tlm_*_get exports on the other side

```
class analysis_fifo #( type T = int ) extends tlm_fifo #( T );

  analysis_imp #( analysis_fifo #( T ) , T ) analysis_export;

  function new( string nm  ,  named_component p = null  );
    super.new( nm , p , 0 );
    analysis_export = new( this );
  endfunction

  function void write( input T t );
    try_put( t );
  endfunction

endclass
```

Analysis fifo *IS A* tlm_fifo

Analysis fifo *IMPLEMENTS* the analysis interface (wrapper)

Analysis fifo is unbounded

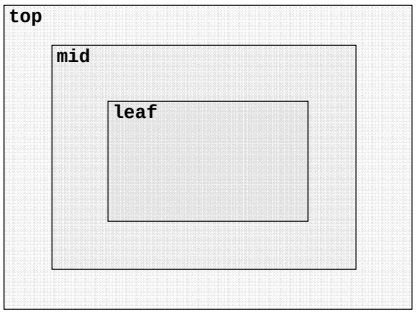
Analysis fifo *HAS A* write method

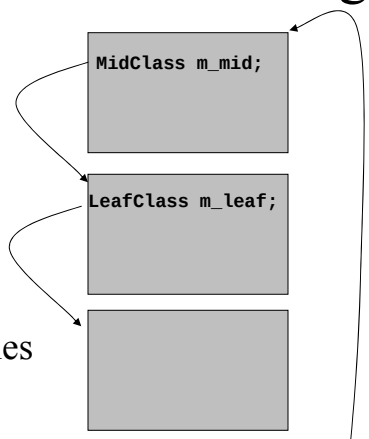
The Advanced Verification Library

AVM Components

- All components are extended from one of two base classes
- `avm_named_component`
  - Maintains name hierarchy based on instantiation
  - Manages hierarchical connections of ports and exports
  - Includes built-in message handling and reporting
- `avm_verification_component`
  - Extended from `avm_named_component`
  - Adds a user-defined `run()` method
  - Includes process control for suspend, resume, kill
  - Defines static `run_all()` method that calls `run()` on every `avm_verification_component`

The Need For Class Instance Naming





- Modules have hierarchical names
  `top;`
  `top.mid;`
  `top.mid.leaf;`
- Classes have handles
  `TopClass m_top;`

avm_named_component

- Base class from which all components are derived
- Builds static associative array of component names
  - Each child appends its name to that of its parent
  - Name for each component passed in via `new()`
- Names are used when issuing messages for components

```
class bot_c extends avm_named_component;
...
endclass

class mid_c extends avm_named_component;
  bot_c b1 = new("b1",this);
  bot_c b2 = new("b2",this);
...
endclass

class top_c extends avm_named_component;
  mid_c m1 = new("m1",this);
  mid_c m2 = new("m2",this);
...
endclass
```

```
class my_env extends avm_env;
  top_c t1 = new("t1");
  top_c t2 = new("t2");
...
endclass
```

avm_named_component

- Provides hierarchical naming throughout the environment
  - Class instance names are essentially invisible
  - HDL modules have an automatic hierarchy of instance names
  - Need to be able to identify and control specific class instances
    - E.g. for messaging and configuration
  - **avm_named_component** provides a mechanism for instance naming
    - Internally, this is a static associative array indexed by string
    - Since it is static, the list of names is shared by all instances
    - Important so that instances can be uniquely identified
- Provides the reporting infrastructure
  - Every **avm_named_component** has its own local message handler
  - Together with the naming facilities, provides per-instance configuration of VIP reporting actions
- Provides access to the connectivity infrastructure
- All verification objects are extended from **avm_named_component**

avm_named_component (Contd.)

- An **avm_named_component** derivative has
  - List of children (which may be empty)
  - A parent (but see **avm_env** for caveat)
  - A *unique* user-defined name
  - Methods to manage connectivity
    - To the children
    - Imports and exports of the TLM interfaces
- There are several methods the user must define
  - Constructor
    - Name and parent are set here
    - Any children should be initialized
  - **connect()**
  - **export_connections()**
  - **import_connections()**
  - **report()** (optional definition)

avm_named_component Constructor

- User must define the constructor of **`avm_named_component`** derivative
  - Also remember to call the parent constructor

`function new(string name, avm_named_component parent = null);`

- **`parent`** is omitted for children of avm_env
- otherwise, **`parent`** is "**`this`**"
- Local instance "**`name`**" must be unique

```
class pipelined_bus_hierarchical_monitor extends avm_named_component;
  address_phase_component m_address_component;
  data_phase_component    m_data_component;

 function new( string name, avm_named_component parent = null );
     super.new( name , parent ); // register name and parent
     m_address_component = new("address_component" , this );
     m_data_component    = new("data_component" , this );
 endfunction

endclass
```

Children

Initialize
set name of childen
set parent of children to "this"

avm_named_component Connection methods

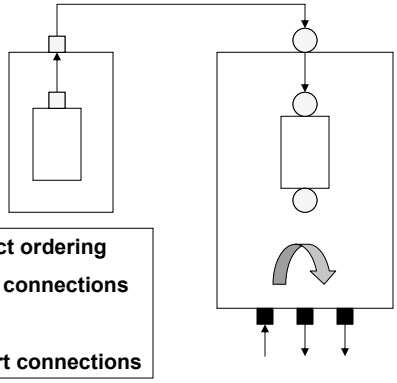
- VIP creator must define the connection methods
  - Three different classes of connections
- `Function void connect()`
  - Connects child ports and exports
- `export_connections()`
  - For interfaces *provided* by this VIP
- `import_connections()`
  - For interfaces *required* by this VIP
- Following transactor example illustrates what is required to be defined

Hierarchical Binding

```
function void connect
  stimulus.blocking_put_port = driver.blocking_put_export;
endfunction
```

```
function void import_connections
  sub_block.blocking_put_port = blocking_put_port;
endfunction
```

```
function void export_connections
  blocking_put_export = fifo.blocking_put_export;
endfunction
```



avm_env ensures correct ordering

1. **Up and Out – export connections**
2. **Across – connect**
3. **Down and In – import connections**

avm_verification_component

- Base class from which all *runnable* components are derived

Extended from avm_named_component

```
virtual class avm_verification_component extends avm_named_component;
  local static avm_verification_component s_component_registry[$];

  process m_main_process;

  function new( string name = "" ,  avm_named_component parent = null ) ;
    super.new( name , parent );
    s_component_registry.push_back( this );
  endfunction

  static task run_all;
    ...; fork s_component_registry[j].m_main_process = process::self;
            s_component_registry[j].run(); join_none ...
  endtask

  static task kill_all;
  ...; s_component_registry[i].m_main_process.kill; ...
  endtask

  virtual task run();
  endtask
endclass
```

Fine-grained process control for run method

Add self to registry

Get process ID for each run method

Invoke all run methods

Kill all run methods

All verification components must define their own run method

AVM Transactors

- AVM includes examples and base classes
  - Users are free to use them as-is or extend them as needed
- avm_stimulus: Generic constrained-random stimulus generator
  - Uses “factory pattern” to create randomized stimulus
  - User is free to modify and/or extend constraints to guide stimulus
- Scoreboarding
  - avm_in_order_comparator
    - Compares two transaction streams of the same type
  - avm_algorithmic_comparator
    - Converts one transaction type to another and then compares
- Drivers, responders, monitors
  - Several protocol-specific examples that can be used or extended

AVM Transactions

- The avm_transaction base class is the basis for all transaction objects
  - Requires the user to define transaction-specific methods called by other components
    - convert2string(): generate a string that gets displayed by print()
    - clone(): defines the action to be performed when copying the transaction (handle-only, shallow copy, deep copy, etc.)
    - comp(): defines which fields are relevant when comparing two transaction
  - clone() and comp() methods are used by other AVM components to handle the transaction correctly

AVM Transaction Class

- **`avm_transaction`** class is a base class
  - AVM and TLM libraries needs it and a few methods
  - Required to print, compare and clone transactions
- Transactions are handles
  - Transactions needs to cloned before they are "sent"

```
typedef enum bit[1:0]  {IDLE, ROCK, PAPER, SCISSORS} rps_t;
class rps_c extends avm_transaction;
  rand rps_t rps;
  function string convert2string;
    return rps.name;
  ...
  function rps_c clone;
    clone = new;
    clone.rps = this.rps;
...
```


AVM Environment Class

- Base avm_env class controls the environment
  - Constructor instantiates and 'new's all components
  - virtual do_test() method controls building and execution
    - Connects all components, virtual interfaces, etc.
    - Configures components and DUT
    - Starts all avm_verification_components
    - Runs the test
    - Reports results
    - Stops everything and exits
  - User defines all application-specific behavior by extending the avm_env class and defining the method bodies

avm_env

```
virtual class avm_env;

virtual task do_test;
    // connect up exports first ( bottom up )
    avm_named_component::export_top_level_connections;

    // then connect "my" children's ports to their siblings' exports
    connect();

    // then propagate port connections down through the
    // hierarchy ( top down )
    avm_named_component::import_top_level_connections;
    configure;

    // execution phases
    avm_verification_component::run_all;
    execute;

    // finish
    report;
    avm_named_component::report_all;
    avm_verification_component::kill_all;
  endtask
```


avm_env(2)

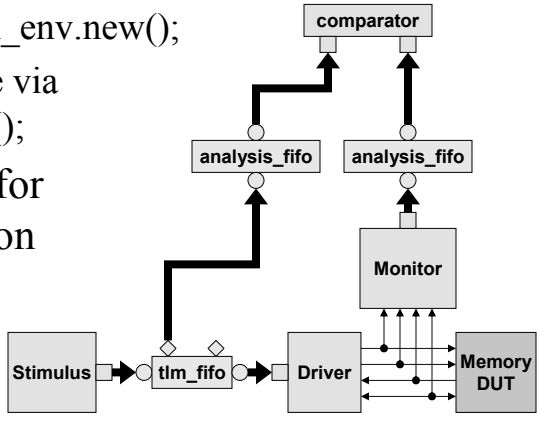
```
  // connect must be overloaded in any subclass. It connects
  // up the top level objects of the testbench.
  pure virtual function void connect;

  // configure is a function - ie no interaction with the
  // scheduler is allowed.
  virtual function void configure;
    return;
  endfunction

  // The execute task is where stimulus generation is
  // started and stopped, and scoreboards and coverage
  // objects are examined from within the testbench, if this
  // is required.
  pure virtual task execute; // execute phase

  // The report function is used to report on the status of
  // the avm_env subclass at the end of simulation.
  virtual function void report;
    avm_report_message("avm_env" , "Finished Test");
    return;
  endfunction

endclass
```


AVM Messaging

- Four levels of Severity:
  MESSAGE, WARNING, ERROR, FATAL
- Four potential actions:
  DISPLAY, LOG, COUNT, EXIT
  - Actions defined by severity, id, or (severity,id) pair
  - Verbosity argument allows for filtering
  - Actions and verbosity can be defined and overridden hierarchically
- Reporting is built into avm_named_component
  - All reporting methods can also be called directly

AVM Example

- All verification components are classes
- Instantiated within an environment class
  - Allocated via avm_env.new();
  - Connections made via avm_env.connect();
- Virtual interfaces for pin-level connection to the DUT

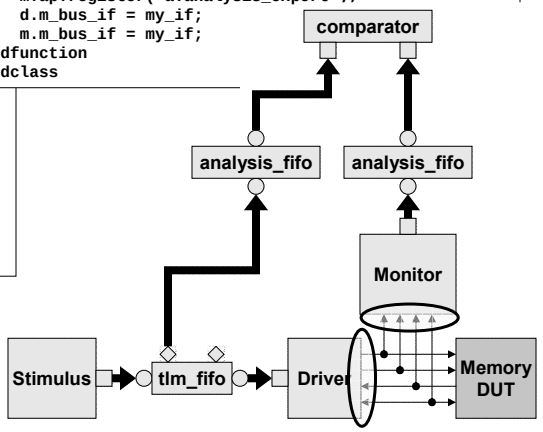


AVM Example

```
class mem_env extends avm_env;
  virtual mem_if my_if;
  stimulus #( trans_type )      s;
  driver #(trans_type)          d;
  monitor #(mem_cov_t)          m;
  comparator #(trans_type,
    avm_class_comp(trans_type)) c;
  tlm_fifo #( trans_type )      f;
  analysis_fifo #( trans_type ) b;
  analysis_fifo #( trans_type ) a;
function new(virtual mem_if bus);
    my_if = bus;
    f = new("fifo");
    b = new("before");
    a = new("after");
    s = new("stimulus");
    d = new("driver");
    m = new("monitor");
    c = new("comparator");
  endfunction
```

```
program mem_tb( mem_if my_if );
  mem_env m_env = new(my_if);
  initial begin
    m_env.connect;
    m_env.run;
  end
endprogram
```

```
function void connect;
    s.blocking_put_port = f.blocking_put_export;
    d.nonblocking_get_port  = f.nonblocking_get_export;
    c.exp_blocking_get_port = b.blocking_get_export;
    c.act_blocking_get_port = a.blocking_get_export;
    f.put_ap.register( b.analysis_export );
    m.ap.register( a.analysis_export );
    d.m_bus_if = my_if;
    m.m_bus_if = my_if;
endfunction
endclass
```



AVM Example: Driver

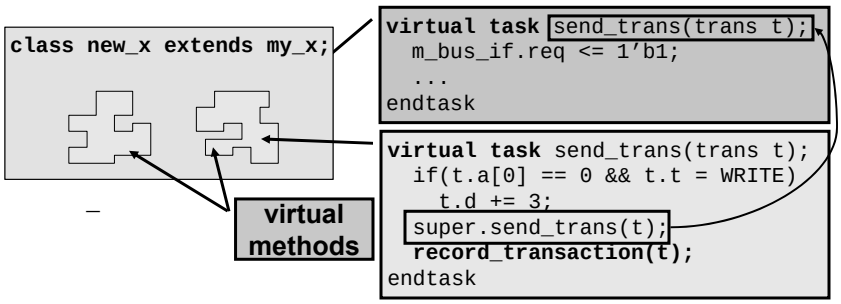
```
class driver #(type T = transaction)extends avm_verification_component;
  typedef enum { READY_TO_SEND_REQ , WAIT_FOR_ACK driver_state;
  tlm_nonblocking_get_if #(T) nonblocking_get_port;
  virtual mem_if m_bus_if;

  local driver_state m_state;  T t;

  task run;
    T t;
    forever begin
      @(posedge m_bus_if.master_mp.clk)
        case( m_state )
          READY_TO_SEND_REQ:
            if( nonblocking_get_port.try_get( t )) begin
              send_to_bus( t );
              m_state <= WAIT_FOR_ACK;
            end
          WAIT_FOR_ACK:begin
            if( m_bus_if.master_mp.ack == 1 ) begin
              m_state <= READY_TO_SEND_REQ;
              master_mp.req <= 0;
            end
        endcase
  endtask
```

Overload to do error injection

```
  protected virtual task send_to_bus( T req );
    m_bus_if.master_mp.req <= 1;
    ...
  endtask
endclass
```

OOP Customization

- This is what Inheritance is *for*



- Modified behavior lives with the component

AVM Example: Error Injection

```
class error_driver #(type T = transaction) extends driver#(T);
  protected virtual task send_to_bus( T req );
    if(req.a[0] == 0 && req.t = WRITE) req.d += 3;
      super.send_to_bus(req);
  endtask
endclass
```

Overload to do error injection

```
class env;
  ...
  error_driver #(trans_type)          d;
  ...
  function void connect;
    ...
    d.nonblocking_get_port = f.nonblocking_get_export;
    ...
  endfunction
endclass
```

Both driver and error_driver have a nonblocking_get_port so the same connection still works

Randomizing Error Injection

```
class rerror_driver #(type T = transaction) extends driver#(T);
  int dly;
  bit par;
  local err_des err = new();

  protected virtual task send_to_bus( T req );
    inject_error( req );
    m_bus_if.master_mp.par <= par;
    m_bus_if.master_mp.req <= 1;
    wait(m_bus_if.master_mp.gnt == 1);
    for(int i=0; i<dly; i++);
      ...
  endtask

  protected virtual task inject_error(ref T req);
    assert(randomize(err));
    case(err.t)
      XDATA:  if(req.a[0] == 0 && req.t = WRITE) req.d += 3;
      DELAY:  assert(randomize(dly) with {dly < err.dly;});
      PARITY: par = !^req.d; // inject parity error
    ...
    endcase
  endtask

endclass
```

```
class err_des;
  rand enum {XDATA, DELAY, PARITY, ...} t;
  rand int dly;
  ...
endclass
```

modify transaction contents and/or set protocol violations

modified procedural code to drive transaction onto bus

randomize error type

modify transaction contents or protocol parameters based on error type

`avm_stimulus` class

- avm_stimulus is a generic stimulus generator
  - Also as prototype for other stimulus generators
- Complete with a put port
  - Connects to channels such as tlm_fifo
- Parameters of sub type avm_transaction
  - One to define the type of transaction sent out across to the put port
  - Optional class containing constraints / used for generation
    - Basically a sub class of the transaction type
- generate_stimulus task generates transactions
  - Base type transactions
  - With optional sub class constraints
  - Uncounted or counted
  - Send transactions to the put port

AVM Stimulus Generator

```
class avm_stimulus #( type trans_type = avm_transaction ) extends
avm_named_component;

  tlm_blocking_put_if #( trans_type ) blocking_put_port;
  local bit m_stop = 0;

  virtual task generate_stimulus( trans_type t = null ,
                                  input int max_count = 0 );
    trans_type temp;
    if( t == null ) t = new;
      for( int i = 0;
      (max_count == 0 || i < max_count) && !m_stop; i++ ) begin
        assert( t.randomize() );
        temp = t.clone();
        blocking_put_port.put( temp );
        end
  endtask

  virtual function void stop_stimulus_generation;
    m_stop = 1;
  endfunction

endclass : avm_stimulus
```


Directing Stimulus(1)

- Extend and constrain base transaction

```
class my_env extends avm_env;
  avm_stimulus #(mem_request) m_stimulus = new();

  class write_request #( int ADDRESS_WIDTH = 8 , int DATA_WIDTH = 8 )
    extends mem_request #( ADDRESS_WIDTH , DATA_WIDTH );

    constraint write_only { this.m_type == MEM_WRITE; }
  endclass // write_request

  write_request #( ADDRESS_WIDTH , DATA_WIDTH ) m_write_gen = new();

  task execute;
    m_stimulus.generate_stimulus( m_write_gen , 10 );
    m_stimulus.generate_stimulus(); // randomize requests
  endtask
```


Directing Stimulus(2)

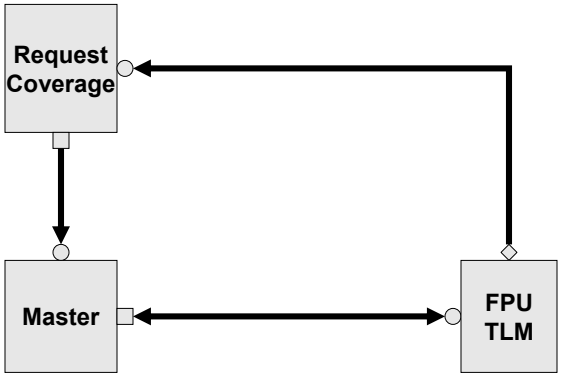
- Define “convenience layer”

```
class my_env extends avm_env;
  class my_stimulus #(type T = mem_request) extends avm_stimulus #(T);
    task write( input address_t address , input data_t data );
      request_t request = new( address , MEM_WRITE , data );
      put_port.put( request );
    endtask
  endclass

  my_stimulus #(mem_request) m_stimulus = new();

  task execute;
    m_stimulus.write( CSR , 16'hA5A5 );
    m_stimulus.write( CLKDIV , 16'h0004 );
    m_stimulus.generate_stimulus(); // randomize requests
  endtask
```

convenience functions for directed stimulus

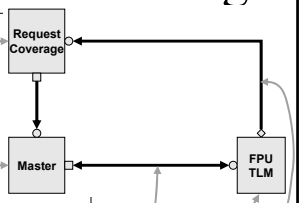
randomized generation from base class

Basic Transaction-Level Testbench



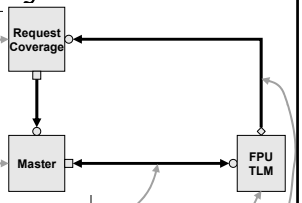
Testbench Environment: SystemVerilog

```
class fpu_env extends avm_env;

  fpu_master  master;
  fpu_tlm     dut;
  fpu_req_cov rqcov;

  function new;
    master = new("master");
    dut = new("fpu_tlm");
    rqcov = new("rqcov");
  endfunction // new

  function void connect();
    master.master_port = dut.blocking_master_export;
    dut.ap.register(rqcov.analysis_export);
  endfunction

  task execute; // execute phase
    fork
      master.go();
      wait_for_coverage();
    join_any
    master.stop();
  endtask

endclass
```

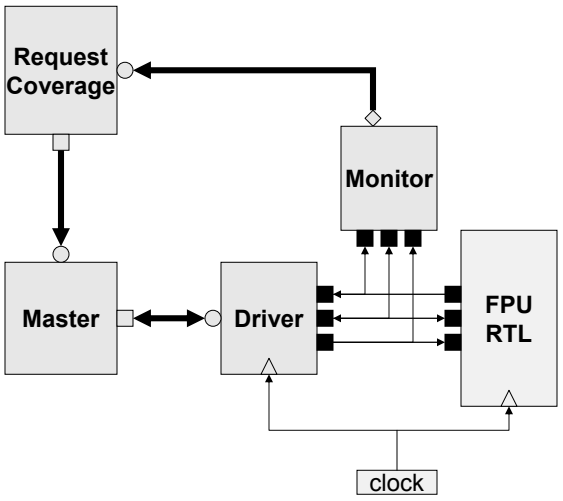


Testbench Environment: SystemC

```
class top : public sc_module
{
 public:

  top(sc_module_name nm) :
      sc_module(nm),
      m("master"),
      dut("fpu_tlm"),
      rq_cov("request_coverage")
  {

    m.master_port(t.master_export);
    t.ap(rq_cov.analysis_export);
  }

  fpu_tlm t;
  fpu_master m;
  fpu_operator_coverpoint rq_cov;
};
```

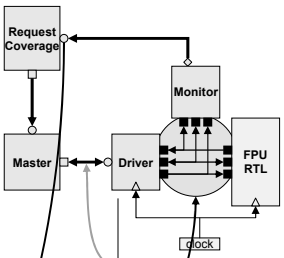


Transaction-Level → RTL Testbench



Transaction-Level → RTL Testbench

```
class fpu_env extends avm_env;

  typedef analysis_port #(fpu_request) rq_ap_t;

  fpu_master  master;
  fpu_driver  driver;
  fpu_req_cov rqcov;
  virtual fpu_pin_if m_fpu_pins;
  rq_ap_t apreq;

  function new(virtual fpu_pin_if pins,
               rq_ap_t apreq);
    master = new("master");
    driver = new("driver");
    rqcov = new("rqcov");
    this.m_fpu_pins = pins;
    this.apreq;
  endfunction // new

  function void connect();
    master.master_port = driver.blocking_master_export;
    apreq.register( rqcov.analysis_export );
    driver.m_fpu_pins = this.m_fpu_pins;
  endfunction

endclass
```



Module-Based Monitor

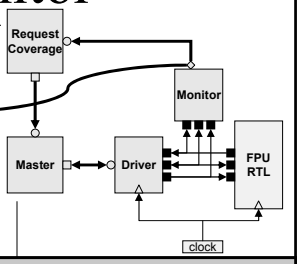
```
module fpu_monitor_mon(interface.monitor_mp pins);
  bit clk;
  analysis_port #(fpu_request) req_ap = new();
  property mon_request;
    logic [31:0] a,b;
    op_t op;
    round_t r;
    @(posedge pins.clk)
      (pins.start, a=pins.opa, b=pins.opb,
                   op=op_t'(pins.fpu_op),
                   r = round_t'(pins.rmode)) |->
       (pins.ready[->1],
        send_rqap(a,b,op,r));
  endproperty
  assert property (mon_request);
endmodule
```

```
function void send_rqap(logic[31:0] a,b,
                        op_t op, round_t r);
  fpu_request req;
  req = new($bitstoshortreal(a),
            $bitstoshortreal(b),
            op, round);
  req_ap.write(req);
endfunction
```

Class parameterized by transaction type

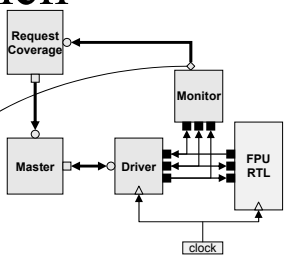
Capture bus signals to local variables within assertion

Send captured transaction to analysis_port



Top-Level Testbench

```
module top;
  bit clk;
  clkgen clkgen(clk); // drive clk
  fpu_pin_if fpu_pins(clk);
  fpu fpu_dut(.clk_i(fpu_pins.clk),
              .opa_i(fpu_pins.opa),...);
  fpu_monitor_mon monitor(fpu_pins);
  fpu_env my_env = new(fpu_pins,
                       monitor.req_ap);
  initial
    my_env.do_test();
endmodule
```

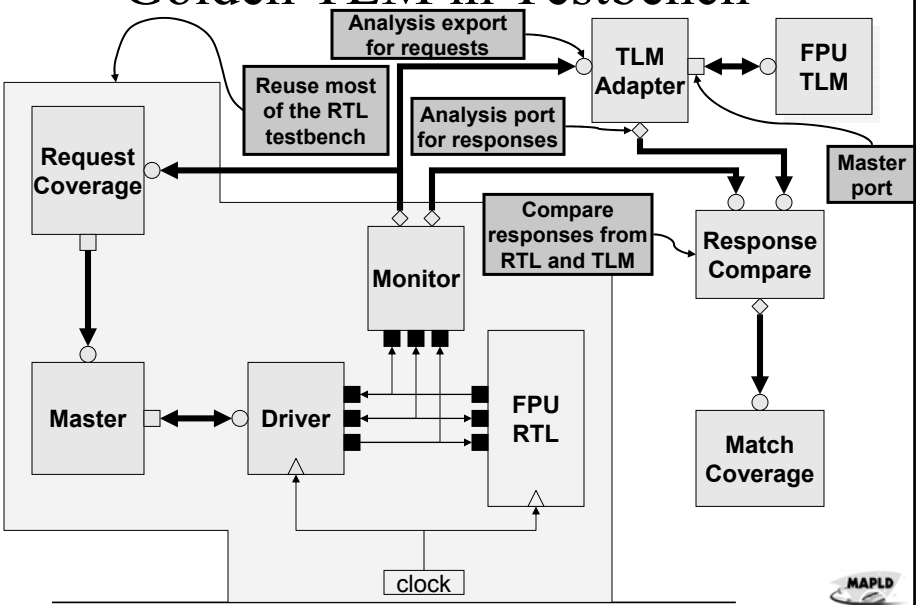


RTL DUT can even be a VHDL model

Monitors and Assertions

- **Module-based monitors are the perfect place to put assertions**
  - **Assertions can automatically check proper DUT behavior on the interface**
  - **Assertions can automatically collect coverage information**
  - **Assertions are not allowed in classes**
- **AVM Module-based monitors communicate with the rest of the testbench via TLM interfaces**
  - **Monitors can be protocol checkers and/or transaction monitors**
  - **Coverage and assertions fully integrated into the testbench**
  - **Can still be used in formal verification**

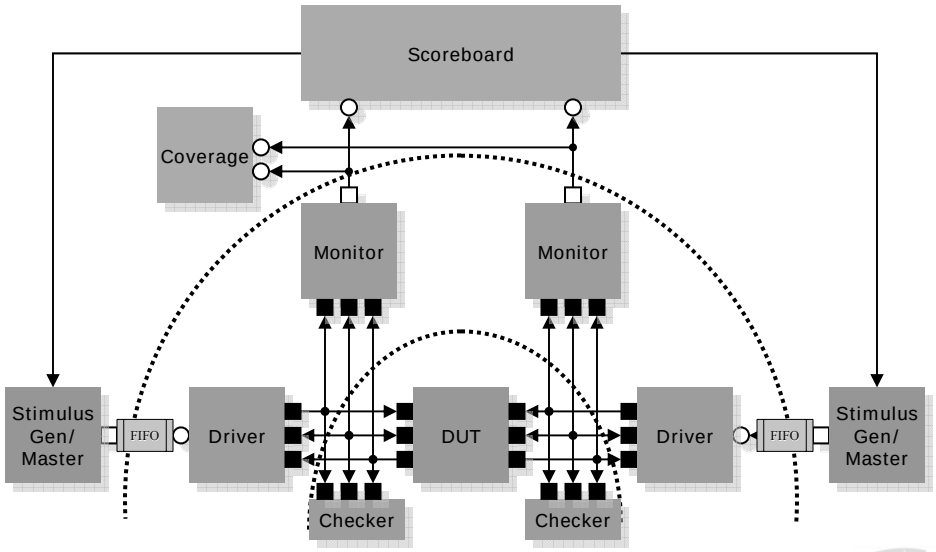
Golden TLM in Testbench



Practical Example

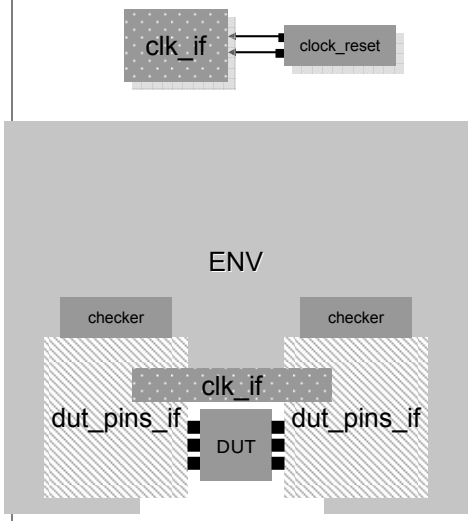
Rock-Paper-Scissors Arbiter

RPS AVM Architecture



RPS – top

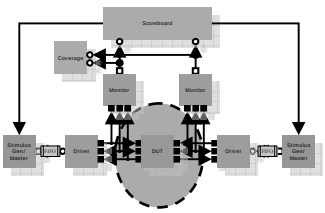
```
module top_class_based;
  import rps_env_pkg::*;

  rps_clk_if      clk_if();
  rps_clock_reset cr(clk_if);

  rps_dut_pins_if pins1_if(clk_if);
  rps_dut_pins_if pins2_if(clk_if);

  rps_dut         dut(pins1_if,
                      pins2_if);
  rps_checker     check1(pins1_if);
  rps_checker     check2(pins2_if);

  rps_env         env;

  initial begin
    env = new(pins1_if,
              pins2_if);
    fork
     cr.run();
    join_none
    env.do_test;
    $finish;
  end
endmodule
```



RPS - DUT

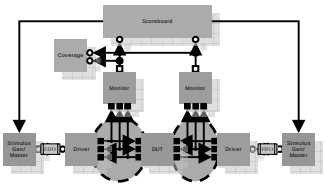
```
module rps_dut(
    rps_dut_pins_if pins1_if, pins2_if);

  bit win1, win2; // Who wins
  int tie_score;  // For debu
  assign
    both_ready =
              (pins1_if.go &
               pins2_if.go);
  initial
      tie_score <= 0;
```

```
  always @(posedge both_ready) begin
    #3; // Consume time
    win1 <= ((pins1_if.r & pins2_if.s) |
             (pins1_if.s & pins2_if.p) |
             (pins1_if.p & pins2_if.r));

    win2 <= ((pins2_if.r & pins1_if.s) |
             (pins2_if.s & pins1_if.p) |
             (pins2_if.p & pins1_if.r));

    if (win1)
      pins1_if.score <= pins1_if.score + 1;
    else if (win2)
      pins2_if.score <= pins2_if.score + 1;
    else
      tie_score <= tie_score + 1;

    pins1_if.clk_if.dut_busy <= 1;
    @(posedge pins1_if.clk_if.clk);
    pins1_if.clk_if.dut_busy <= 0;
  end
endmodule
```



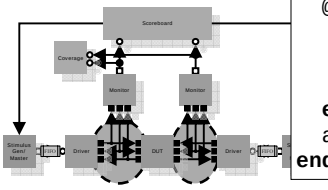
RPS – DUT Pins Interface

```
interface rps_dut_pins_if(rps_clk_if clk_if);
  import rps_env_pkg::*;

  reg r, p, s; // Rock, Paper, Scissors.
               //   Mutually exclusive.
  int score;   // # of times THIS player has won.
  reg go;      // Posedge -> the DUT should
               //   start calculating.

  rps_t play;  // Enum used only in debug.

endinterface
```



RPS – DUT Protocol Checkers

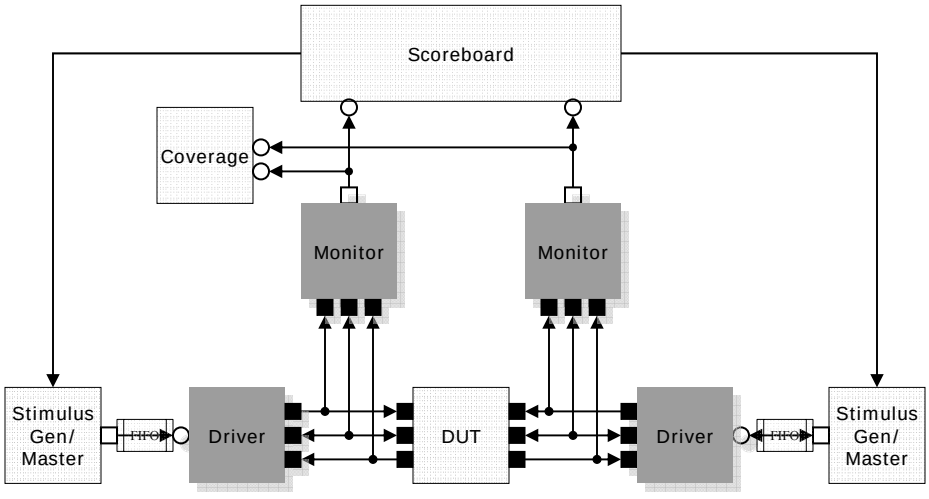
```
module rps_checker(rps_dut_pins_if pins_if);
  // 1. RPS and score are never unknown on
  //    posedge of clk.
  property no_meta;
  @(posedge clk_if.clk) disable iff (clk_if.rst)
    pins_if.go |=>
      $isunknown({pins_if.r,pins_if.s,pins_if.p,
        pins_if.score}) == 0;
  endproperty
  assert_no_meta: assert property (no_meta);

  // 2. Only one of RPS is 1
  property valid_play;
  @(posedge clk_if.clk) disable iff (clk_if.rst)
    pins_if.go |=>
      $countones({pins_if.r,pins_if.s,pins_if.p})
        == 1;
  endproperty
  assert_valid_play: assert property (valid_play);
endmodule
```



RPS - Transaction

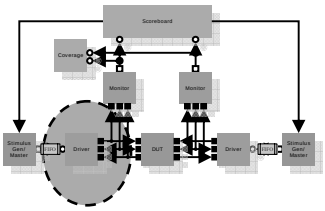
```
typedef enum bit[1:0] {IDLE, ROCK, PAPER, SCISSORS} rps_t;

class rps_c extends avm_transaction;
  rand rps_t rps;
  constraint illegal { rps != IDLE; }
  int score;

  function string convert2string;
    return rps.name;
  endfunction

  function bit comp(input rps_c a);
    if (a.rps == this.rps)
      return 1;
    else
      return 0;
  endfunction

  function rps_c clone;
    clone = new;
    clone.rps = this.rps;
  endfunction
endclass
```


RPS - Transactors



RPS - Driver

```
class rps_driver
    extends avm_verification_component;

  tlm_nonblocking_get_if #(rps_c) nb_get_port;
  rps_c transaction;

  virtual rps_dut_pins_if pins_if;

  function new(string nm = "",
      avm_named_component p = null);
    super.new(nm, p);
  endfunction
```

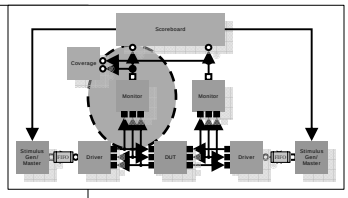


RPS - Driver

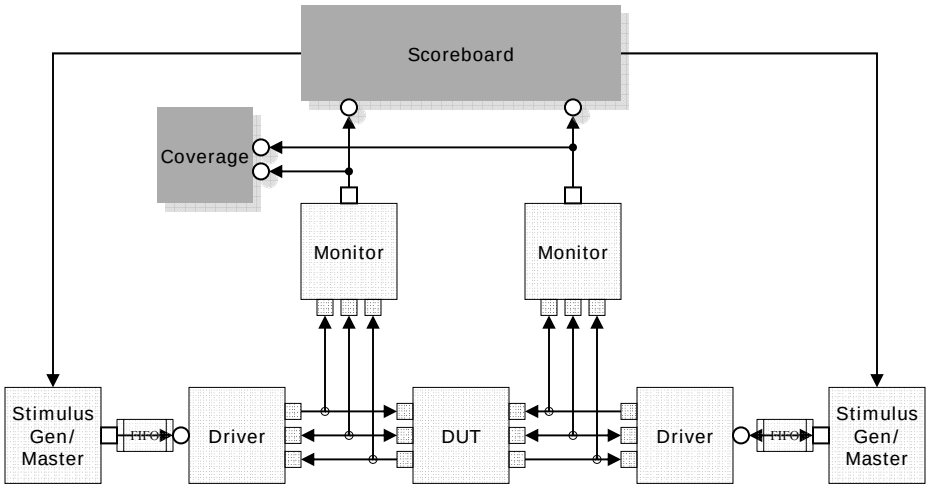
```
  task run;
    {pins_if.r, pins_if.p, pins_if.s} = 3'b000;
    pins_if.go = 0;
    @(negedge pins_if.clk_if.rst);
    forever @(posedge pins_if.clk_if.clk)
      if (nb_get_port.try_get(transaction))
      begin
        pins_if.play = transaction.rps; //Debug.
        {pins_if.r, pins_if.p, pins_if.s} = 3'b000;
        case (transaction.rps)
              ROCK: pins_if.r = 1;
             PAPER: pins_if.p = 1;
          SCISSORS: pins_if.s = 1;
        endcase
        pins_if.go = 1;
        @(posedge pins_if.clk_if.clk);
        pins_if.go = 0;
        @(posedge pins_if.clk_if.clk);
        {pins_if.r, pins_if.p, pins_if.s} = 3'b000;
      end
  endtask
endclass
```


RPS - Monitor

```
class rps_monitor
     extends avm_verification_component;
   virtual rps_dut_pins_if pins_if;
   analysis_port #(rps_c) ap;
   function new(string nm = "",
       avm_named_component p = null);
     super.new(nm, p);
     ap = new;
   endfunction
   function rps_c pins2transaction;
     rps_c transaction = new;
     case ({pins_if.r,pins_if.p,pins_if.s})
       3'b100: transaction.rps = ROCK;
       3'b010: transaction.rps = PAPER;
       3'b001: transaction.rps = SCISSORS;
     endcase
     transaction.score = pins_if.score;
     return transaction;
   endfunction
   task run;
      forever @(posedge pins_if.clk_if.dut_busy)
       ap.write(pins2transaction());
   endtask
 endclass
```

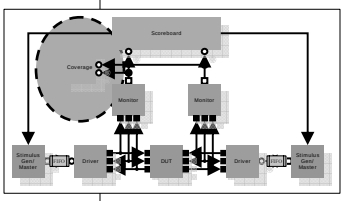


RPS - Coverage & Scoreboard



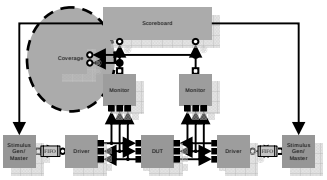
RPS - Coverage

```
class rps_coverage
     extends avm_verification_component;

   local analysis_fifo #(rps_c) fifo1, fifo2;
   analysis_if         #(rps_c) analysis_export1,
                                analysis_export2;

   rps_c t1, t2;
   rps_t t1rps, t2rps;

   covergroup rps_cover;
     coverpoint t1rps {
       ignore_bins illegal = { IDLE };
     }
     coverpoint t2rps {
       ignore_bins illegal = { IDLE };
     }
     cross t1rps, t2rps;
   endgroup

   function void export_connections;
     analysis_export1 = fifo1.analysis_export;
     analysis_export2 = fifo2.analysis_export;
   endfunction
```



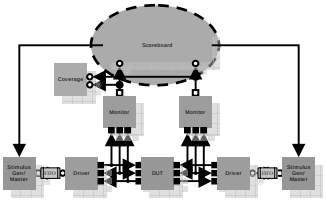
RPS - Coverage

```
   function new(string nm = "",
       avm_named_component p = null);
     super.new(nm, p);
     fifo1 = new("fifo1", this);
     fifo2 = new("fifo2", this);
     rps_cover = new;
   endfunction

   task run;
     forever begin
       fifo1.get(t1);
       fifo2.get(t2);
       report_the_play("COV", t1, t2);
       t1rps = t1.rps;
       t2rps = t2.rps;
       rps_cover.sample;
     end
   endtask
 endclass
```



RPS - Scoreboard

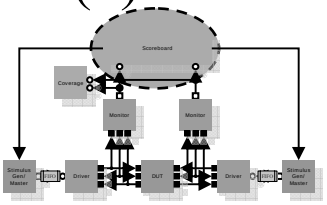
```
class rps_scoreboard
    extends avm_verification_component;

  local analysis_fifo #(rps_c) fifo1, fifo2;
  analysis_if         #(rps_c) analysis_export1,
                               analysis_export2;

  rps_c t1, t2;
  int score1, score2, tie_score;

  int limit;     //Gets set at configure time.
  reg test_done; //Gets waited on externally.

  function new(string nm = "",
      avm_named_component p = null);
    super.new(nm, p);

    fifo1 = new("fifo1", this);
    fifo2 = new("fifo2", this);
    test_done = 0;
    score1 = 0; score2 = 0;
    tie_score = 0;
  endfunction
```




```
  function void export_connections;
    analysis_export1 = fifo1.analysis_export;
    analysis_export2 = fifo2.analysis_export;
  endfunction

  task run;
    forever begin
      fifo1.get(t1);
      fifo2.get(t2);
      report_the_play("SBD", t1, t2);
      update_and_check_score();
    end
  endtask
```


RPS – Scoreboard(2)

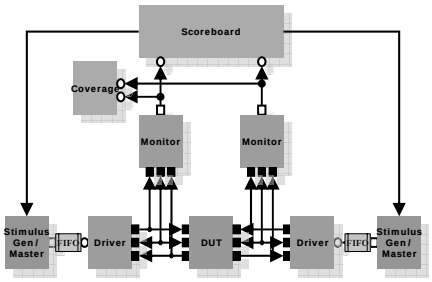
```
local function void update_and_check_score;
  string str;
  bit win1, win2;

  // Validate the score.
  if (score1 != t1.score) begin
    $sformat(str,
      "MISMATCH - score1=%0d, t1.score=%0d",
      score1, t1.score);
    avm_report_message("SBD", str);
    end
  if (score2 != t2.score) begin
    $sformat(str,
      "MISMATCH - score2=%0d, t2.score=%0d",
      score2, t2.score);
    avm_report_message("SBD", str);
    end

  // Calculate new score.
  win1 =
   ((t1.rps == ROCK     && t2.rps == SCISSORS) |
    (t1.rps == SCISSORS && t2.rps == PAPER)    |
    (t1.rps == PAPER    && t2.rps == ROCK));

  win2 =
   ((t2.rps == ROCK     && t1.rps == SCISSORS) |
    (t2.rps == SCISSORS && t1.rps == PAPER)    |
    (t2.rps == PAPER    && t1.rps == ROCK));
```




```
    if (win1)
      score1 += 1;
    else if (win2)
      score2 += 1;
    else
      tie_score +=1;

   // Check for done.
    if ((t1.score >= limit) ||
        (t2.score >= limit))
      test_done = 1;

  endfunction
endclass
```

RPS – rps_env

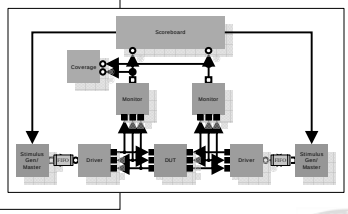
```
class rps_env extends avm_env;
   local virtual
     rps_dut_pins_if m_pins1_if, m_pins2_if;

   avm_stimulus #(rps_c) s1, s2;
   tlm_fifo     #(rps_c) f1, f2;
   rps_driver            d1, d2;
   rps_monitor           m1, m2;
   rps_scoreboard        sb;
   rps_coverage          c;
```




```
   function new(
    virtual rps_dut_pins_if pins1_if,
                            pins2_if);

    s1 = new("rps_stimulus1");
    f1 = new("rps_fifo1");
    d1 = new("rps_driver1");

    s2 = new("rps_stimulus2");
    f2 = new("rps_fifo2");
    d2 = new("rps_driver2");

    m1 = new("rps_monitor1");
    m2 = new("rps_monitor2");

    sb = new("rps_scoreboard");
    c  = new("rps_coverage");

    m_pins1_if = pins1_if;
    m_pins2_if = pins2_if;
   endfunction
```

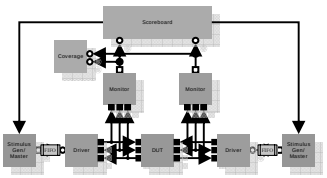

RPS – rps_env

```
   function void connect;
     // Hook up 1
     s1.blocking_put_port = f1.blocking_put_export;
     d1.nb_get_port       = f1.nonblocking_get_export;

     d1.pins_if     = m_pins1_if;
     m1.pins_if     = m_pins1_if;

     m1.ap.register(sb.analysis_export1);
     m1.ap.register(c.analysis_export1);

     // Hook up 2
     s2.blocking_put_port = f2.blocking_put_export;
     d2.nb_get_port       = f2.nonblocking_get_export;

     d2.pins_if     = m_pins2_if;
     m2.pins_if     = m_pins2_if;

     m2.ap.register(sb.analysis_export2);
     m2.ap.register(c.analysis_export2);
   endfunction

   function void configure;
     sb.limit = 10;
   endfunction
```



RPS – rps_env

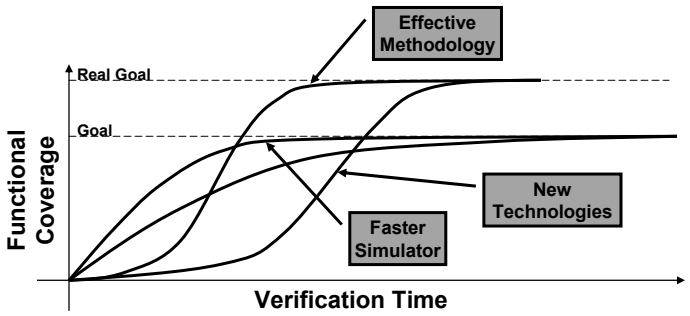
```
task execute;
  fork
    s1.generate_stimulus();
    s2.generate_stimulus();
    terminate;
  join
endtask

task terminate;
  @(posedge sb.test_done);
  s1.stop_stimulus_generation();;
  s2.stop_stimulus_generation();;
endtask

function void report;
  string str;
  $sformat(str, "%d %% covered",
      c.rps_cover.get_inst_coverage());
  avm_report_message("coverage report",
    str);
endfunction
```



Summary

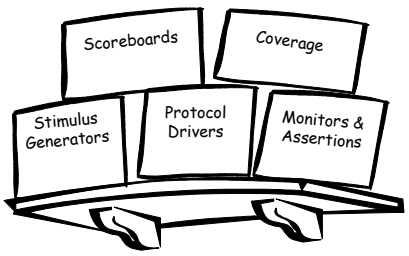
Verification Productivity

- Productivity means better coverage in less time
- Faster simulation = same coverage in less time
- New technologies help find more bugs
  - Functional Coverage, Testbench Automation, Assertions
- Focused methodology applies these technologies more effectively



The Advanced Verification Methodology (AVM)

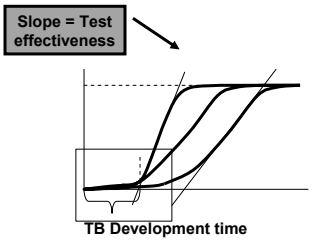
- Purpose:
  - To help users build verification environments to take advantage of our technology
- The Ingredients:
  - A library of modular, reusable Verification Components
  - Examples
  - Documentation



- Infrastructure details built-in so you don't have to worry about them
- Open-Source means you are free to use them, with full access to all the source code

AVM Value Statement

- Shorten the development time of the TB infrastructure
  - The *Verification Cookbook* shows you how to do this
  - Verification IP
  - Coding guidelines
  - Multiple abstraction levels
- Improve the effectiveness of tests at verifying features
  - CR simulation (incl. solver)
  - Functional coverage
  - ABV, Formal Model Checking
  - Dynamic Formal



AVM ROI

Assumptions:

Gates/loc	Bugs/100loc	Bugs/M gates	Debug time	Labor
10	**.75**	**750**	**.9 man-days/bug**	**$800/man-day $16,000/man-month**

Find bugs sooner

Actual Results:

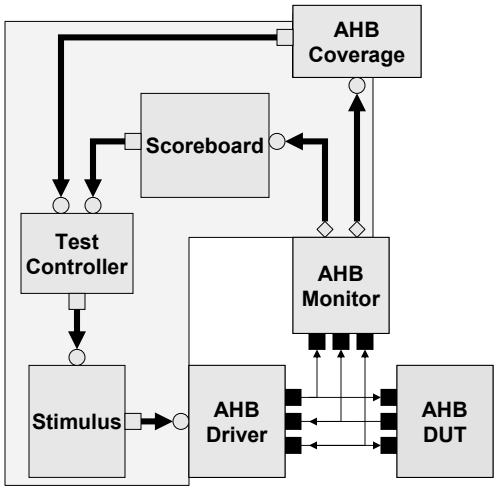
	Description	Factor	Example	Impact	Savings
ABV	**Debug savings**	**25% avg. reduction**	**3000 bugs/ 4M gates**	**3000 x .25 x .9=** 675 man-days	34 man-months
TLM	**Bug reduction**	**.07 design & verification**	**3000 x .07 ≈ 200 bugs/4M gates**	**(3000 – 200) x .9 =** 2520 m-d	63 man-months
TLM TB	**Productivity**	**2x**	**32 man-months (Verilog directed tests)**	**32/2 =**	16 man-months
TBA	**CR test generation**	**5x**	**3.2 man-months 10 engineers**	**(16-3.2) x 10 =**	128 man-months

100% Coverage

AVM ROI Summary

- Assertion-Based Verification
  - Find bugs earlier; Fix bugs faster
  - 34m-m * $16k = $540,000
- Transaction Level Modeling
  - Reduce verification time; Improve Reuse
  - 79m-m * $16K = $1,264,000
- Testbench Automation
  - Improve verification productivity
  - 128m-m * $16K = $2,048,000
- Reinvest savings to improve quality and completeness
- **Being able to sleep the night you tape-out**
  - **Priceless**

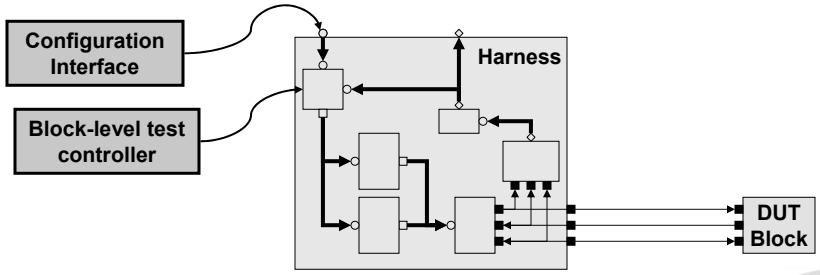
Reuse Across Projects



- **Testbench topology is application-specific**
  - **Reuse TLM components**
  - **Same connections between TLM components**
- **Transactors are protocol-specific**
  - **Scoreboard tracks correct behavior**
  - **Coverage tracks protocol corner-cases**
- **Components can easily be swapped since they use the same interfaces**

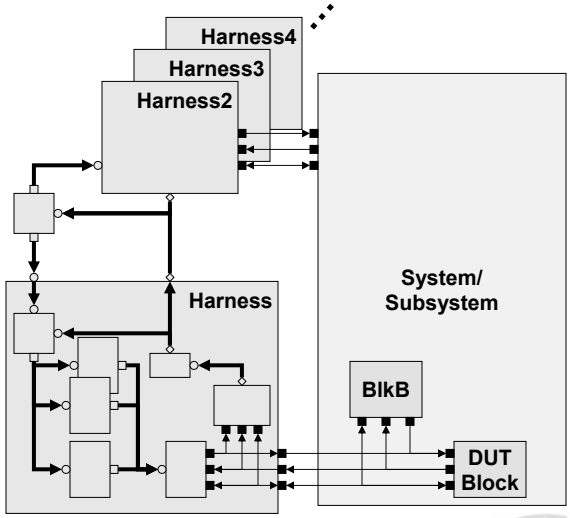
Reuse Within Projects

- Reuse block-level TB in system TB
- Gather block-level TB into a "harness"
  - Contains all necessary block-level verification components
- Extend harness as necessary to communicate with system-level testbench

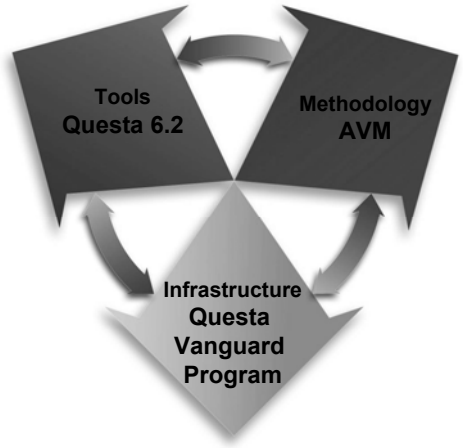


"Block-to-Top" Reuse

- System-Level testbench coordinates behavior of block-level harnesses
- Configuration interfaces customize behavior of each harness
  - How many and what components to instantiate
  - Turn components on/off; active/passive



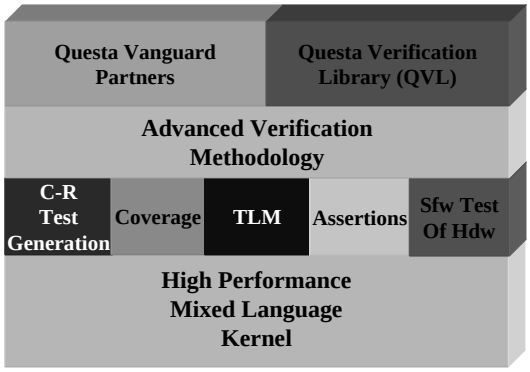
Questa Verification Advances on Three Fronts



Need All Three to Accelerate Adoption of New Flows

Questa Summary

Mentor's Functional Verification Platform



Reusable Verification IP for Productivity

- Mentor's solutions have proven ROI in real designs

Questa Vanguard Program

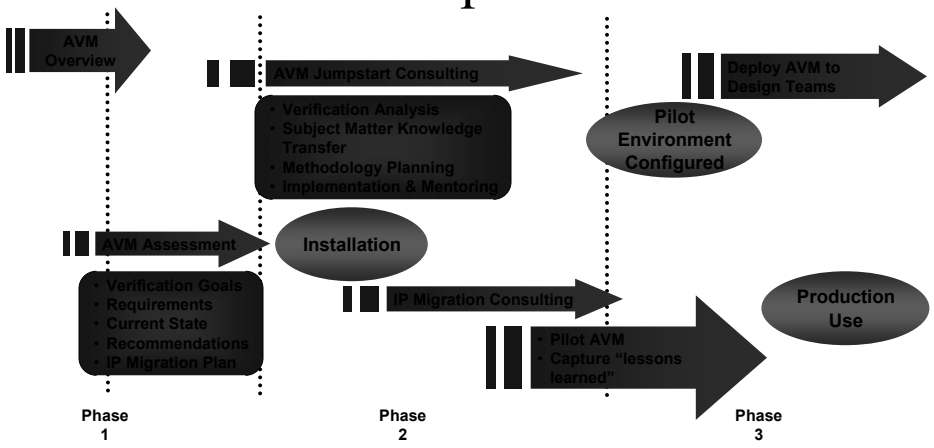
- **A group of well established companies from around the world working with Mentor Graphics to accelerate the adoption of advanced verification techniques and methodologies**
- **QVP Partners offer**
  - **Integrations into Questa**
  - **Training**
  - **Verification IP**
  - **Consulting, Conversion and Migration services**
- **QVP Partners all support Questa and the AVM**

Questa Vanguard Program

Company Listing (July 2006)

- Ace Verification
- ARM Ltd
- Averant
- Doulos
- Denali Software
- eInfoChips
- Expert I/O
- HDL Design House
- hd Lab
- Interra Systems
- Intrinsix
- Mu Electronics
- Nobug
- Novas
- nSys
- PSI Electronics
- Paradigm Works
- Real Intent
- Scaleo Chip
- SiMantis
- Sonics
- SpiraTech
- Sunburst Design
- Sutherland HDL
- Summit
- SyoSil
- TrustIC Design
- VeriEZ Solutions
- Vericine
- Verilab
- Vhdl Cohen Publishing
- XtremeEDA
- Willamette HDL

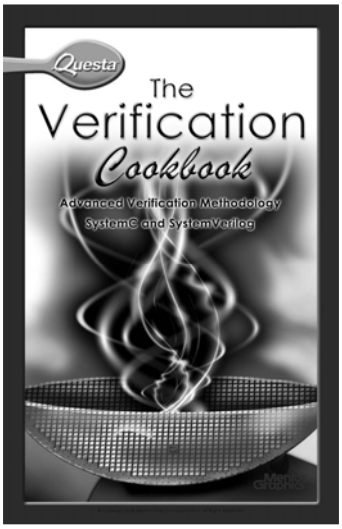
Advanced Verification Methodology Adoption



The Verification Cookbook

- The Recipes you need to know
- Introduces Advanced Verification topics incrementally
- Use or modify the examples provided as needed
- To download the AVM Cookbook please go to

  www.mentor.com/go/cookbook



Questions?

```
//<>================================================<>
//    solution08-rps/t.sv
//<>================================================<>

// $Id: t.sv,v 1.1 2006/09/01 16:09:50 redelman Exp $
//------------------------------------------------------
//   Copyright 2005-2006 Mentor Graphics Corporation
//
//   Licensed under the Apache License, Version 2.0
//   (the "License"); you may not use this file except
//   in compliance with the License.  You may obtain a
//   copy of the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in
//   writing, software distributed under the License is
//   distributed on an "AS IS" BASIS, WITHOUT
//   WARRANTIES OR CONDITIONS OF ANY KIND, either
//   express or implied.  See the License for the
//   specific language governing permissions and
//   limitations under the License.
//------------------------------------------------------

/*
 * rps-class-based - Rock, Paper, Scissors Example.
 * Based on code created by David Jones at Xtreme EDA
 * for an article in Verification Horizons Q4 2005 -
 * Vol. 1, Issue 1, entitled "A Reusable SystemVerilog
 * Testbench in Only 300 Lines of Code".
 */

interface rps_clk_if;
  bit clk, rst, dut_busy;
endinterface

package rps_env_pkg;
  import avm_pkg::*;

  typedef enum bit[1:0]
    {IDLE, ROCK, PAPER, SCISSORS} rps_t;

  class rps_c extends avm_transaction;
    rand rps_t rps;
    constraint illegal { rps != IDLE; }
    int score;

    function string convert2string;
      return rps.name;
    endfunction

    function bit comp(input rps_c a);
      if (a.rps == this.rps)
        return 1;
      else
        return 0;
    endfunction

    function rps_c clone;
      clone = new;
      clone.rps = this.rps;
    endfunction
  endclass

  function void report_the_play(
      string where, rps_c t1, rps_c t2);
    string str;
    $sformat(str,
        "(%s, %s) - Score1=%0d, Score2=%0d",
        t1.rps.name, t2.rps.name,
        t1.score, t2.score);
    avm_report_message(where, str);
  endfunction

  class rps_coverage
      extends avm_verification_component;

    local analysis_fifo #(rps_c) fifo1, fifo2;
    analysis_if         #(rps_c) analysis_export1,
                                 analysis_export2;

    rps_c t1, t2;
    rps_t t1rps, t2rps;

    covergroup rps_cover;
      coverpoint t1rps {
        ignore_bins illegal = { IDLE };
      }
      coverpoint t2rps {
        ignore_bins illegal = { IDLE };
      }
      cross t1rps, t2rps;
    endgroup

    function void export_connections;
      analysis_export1 = fifo1.analysis_export;
      analysis_export2 = fifo2.analysis_export;
    endfunction

    function new(string nm = "",
        avm_named_component p = null);
      super.new(nm, p);
      fifo1 = new("fifo1", this);
      fifo2 = new("fifo2", this);
      rps_cover = new;
    endfunction

    task run;
      forever begin
        fifo1.get(t1);
        fifo2.get(t2);
        report_the_play("COV", t1, t2);
        t1rps = t1.rps;
        t2rps = t2.rps;
        rps_cover.sample;
      end
    endtask
  endclass


  class rps_scoreboard
      extends avm_verification_component;

    local analysis_fifo #(rps_c) fifo1, fifo2;
    analysis_if         #(rps_c) analysis_export1,
                                 analysis_export2;

    rps_c t1, t2;
    int score1, score2, tie_score;

    int limit;     //Gets set at configure time.
    reg test_done; //Gets waited on externally.

    function new(string nm = "",
        avm_named_component p = null);
      super.new(nm, p);

      fifo1 = new("fifo1", this);
      fifo2 = new("fifo2", this);
      test_done = 0;
      score1 = 0; score2 = 0; tie_score = 0;
    endfunction

    function void export_connections;
      analysis_export1 = fifo1.analysis_export;
      analysis_export2 = fifo2.analysis_export;
    endfunction

    task run;
      forever begin
        fifo1.get(t1);
        fifo2.get(t2);
        report_the_play("SBD", t1, t2);
        update_and_check_score();
      end
    endtask

    local function void update_and_check_score;

      string str;
      bit win1, win2;

      // Validate the score.
      if (score1 != t1.score) begin
          $sformat(str,
            "MISMATCH - score1=%0d, t1.score=%0d",
            score1, t1.score);
        avm_report_message("SBD", str);
      end
      if (score2 != t2.score) begin
        $sformat(str,
            "MISMATCH - score2=%0d, t2.score=%0d",
            score2, t2.score);
        avm_report_message("SBD", str);
      end

      // Calculate new score.
      win1 =
     ((t1.rps == ROCK     && t2.rps == SCISSORS) |
      (t1.rps == SCISSORS && t2.rps == PAPER)    |
      (t1.rps == PAPER    && t2.rps == ROCK));

      win2 =
     ((t2.rps == ROCK     && t1.rps == SCISSORS) |
      (t2.rps == SCISSORS && t1.rps == PAPER)    |
      (t2.rps == PAPER    && t1.rps == ROCK));
```

```
    if (win1)
      score1 += 1;
    else if (win2)
      score2 += 1;
    else
      tie_score +=1;

    // Check to see if we are done.
    if ((t1.score >= limit) ||
        (t2.score >= limit))
      test_done = 1;

  endfunction
endclass


class rps_driver
    extends avm_verification_component;

  tlm_nonblocking_get_if #(rps_c) nb_get_port;
  rps_c transaction;

  virtual rps_dut_pins_if pins_if;

  function new(string nm = "",
      avm_named_component p = null);
    super.new(nm, p);
  endfunction

  task run;
    {pins_if.r, pins_if.p, pins_if.s} = 3'b000;
    pins_if.go = 0;
    @(negedge pins_if.clk_if.rst);
    forever @(posedge pins_if.clk_if.clk)
      if (nb_get_port.try_get(transaction))
      begin
        pins_if.play = transaction.rps; //Debug.
        {pins_if.r, pins_if.p, pins_if.s}
          = 3'b000;
        case (transaction.rps)
              ROCK: pins_if.r = 1;
             PAPER: pins_if.p = 1;
          SCISSORS: pins_if.s = 1;
        endcase
        pins_if.go = 1;
        @(posedge pins_if.clk_if.clk);
        pins_if.go = 0;
        @(posedge pins_if.clk_if.clk);
        {pins_if.r, pins_if.p, pins_if.s}
          = 3'b000;
      end
  endtask
endclass


class rps_monitor
    extends avm_verification_component;

  virtual rps_dut_pins_if pins_if;

  analysis_port #(rps_c) ap;

  function new(string nm = "",
      avm_named_component p = null);
    super.new(nm, p);
    ap = new;
  endfunction

  function rps_c pins2transaction;
    rps_c transaction = new;
    case ({pins_if.r,pins_if.p,pins_if.s})
      3'b100: transaction.rps = ROCK;
      3'b010: transaction.rps = PAPER;
      3'b001: transaction.rps = SCISSORS;
    endcase
    transaction.score = pins_if.score;
    return transaction;
  endfunction

  task run;
    forever @(posedge pins_if.clk_if.dut_busy)
      ap.write(pins2transaction());
  endtask
endclass


class rps_env extends avm_env;
  local virtual
    rps_dut_pins_if m_pins1_if, m_pins2_if;

  avm_stimulus #(rps_c) s1, s2;
  tlm_fifo     #(rps_c) f1, f2;
  rps_driver            d1, d2;
  rps_monitor           m1, m2;
  rps_scoreboard        sb;
  rps_coverage          c;

  function void connect;
    // Hook up 1
    s1.blocking_put_port
                  = f1.blocking_put_export;
    d1.nb_get_port = f1.nonblocking_get_export;

    d1.pins_if     = m_pins1_if;
    m1.pins_if     = m_pins1_if;

    m1.ap.register(sb.analysis_export1);
    m1.ap.register(c.analysis_export1);

    // Hook up 2
    s2.blocking_put_port
                  = f2.blocking_put_export;
    d2.nb_get_port = f2.nonblocking_get_export;

    d2.pins_if     = m_pins2_if;
    m2.pins_if     = m_pins2_if;

    m2.ap.register(sb.analysis_export2);
    m2.ap.register(c.analysis_export2);
  endfunction

  function void configure;
    sb.limit = 10;
  endfunction

  task execute;
    fork
      s1.generate_stimulus();
      s2.generate_stimulus();
      terminate;
    join
  endtask

  task terminate;
    @(posedge sb.test_done);
    s1.stop_stimulus_generation();;
    s2.stop_stimulus_generation();;
  endtask

  function void report;
    string str;
    $sformat(str, "%d %% covered",
        c.rps_cover.get_inst_coverage());
    avm_report_message("coverage report", str);
  endfunction

  function new(
    virtual rps_dut_pins_if pins1_if, pins2_if);

    s1 = new("rps_stimulus1");
    f1 = new("rps_fifo1");
    d1 = new("rps_driver1");

    s2 = new("rps_stimulus2");
    f2 = new("rps_fifo2");
    d2 = new("rps_driver2");

    m1 = new("rps_monitor1");
    m2 = new("rps_monitor2");

    sb = new("rps_scoreboard");
    c  = new("rps_coverage");

    m_pins1_if = pins1_if;
    m_pins2_if = pins2_if;
  endfunction
endclass
endpackage

// ----------------------------------------------

module rps_clock_reset(interface i );
  parameter bit ACTIVE_RESET = 1;

  task run(
    int reset_hold = 4 ,
    int half_period = 10 ,
    int count = 0 );

    i.clk = 0;
    i.rst = ACTIVE_RESET;

    for(int rst_i = 0;
        rst_i < reset_hold; rst_i++ ) begin
      #half_period; i.clk = !i.clk;
      #half_period; i.clk = !i.clk;
    end
```

```
    i.rst <= !i.rst;
    // Run the clock
    for(int clk_i = 0;
      (clk_i < count || count == 0);
      clk_i++ ) begin
      #half_period; i.clk = !i.clk;
    end
  endtask
endmodule


interface rps_dut_pins_if(rps_clk_if clk_if);
  import rps_env_pkg::*;

  reg r, p, s; // Rock, Paper, Scissors.
               //   Mutually exclusive.
  int score;   // # of times THIS player has won.
  reg go;      // Posedge -> the DUT should
               //   start calculating.

  rps_t play; // Enum used only in debug.
endinterface

module rps_checker(rps_dut_pins_if pins_if);

  // ****************************************
  // Assertions *****************************
  // ****************************************
  // 1. RPS and score are never unknown on
  //    posedge of clk.
  property no_meta;
  @(posedge clk_if.clk) disable iff (clk_if.rst)
    pins_if.go |=>
      $isunknown({pins_if.r,pins_if.s,pins_if.p,
        pins_if.score}) == 0;
  endproperty
  assert_no_meta: assert property (no_meta);

  // ****************************************
  // 2. Only one of RPS is 1
  property valid_play;
  @(posedge clk_if.clk) disable iff (clk_if.rst)
    pins_if.go |=>
      $countones({pins_if.r,pins_if.s,pins_if.p})
        == 1;
  endproperty
  assert_valid_play: assert property (valid_play);
  // ****************************************
endmodule


module rps_dut(
    rps_dut_pins_if pins1_if, pins2_if);

  bit win1, win2; // Who wins? 1 or 2?
  int tie_score;  // For debug and reporting.

  assign both_ready = (pins1_if.go & pins2_if.go);

  initial
      tie_score <= 0;

  always @(posedge both_ready) begin

    #3; // Consume time
    win1 <= ((pins1_if.r & pins2_if.s) |
             (pins1_if.s & pins2_if.p) |
             (pins1_if.p & pins2_if.r));

    win2 <= ((pins2_if.r & pins1_if.s) |
             (pins2_if.s & pins1_if.p) |
             (pins2_if.p & pins1_if.r));

    if (win1)
      pins1_if.score <= pins1_if.score + 1;
    else if (win2)
      pins2_if.score <= pins2_if.score + 1;
    else
      tie_score <= tie_score + 1;

    pins1_if.clk_if.dut_busy <= 1;
    @(posedge pins1_if.clk_if.clk);
    pins1_if.clk_if.dut_busy <= 0;
  end
endmodule


module top_class_based;
  import rps_env_pkg::*;

  rps_clk_if      clk_if();
  rps_clock_reset cr(clk_if);

  rps_dut_pins_if pins1_if(clk_if);
  rps_dut_pins_if pins2_if(clk_if);

  rps_dut         dut(pins1_if, pins2_if);
  rps_checker     checker1(pins1_if);
  rps_checker     checker2(pins2_if);

  rps_env         env;

  initial begin
    env = new(pins1_if, pins2_if);
    fork
     cr.run();
    join_none
    env.do_test;
    $finish;
  end
endmodule
```