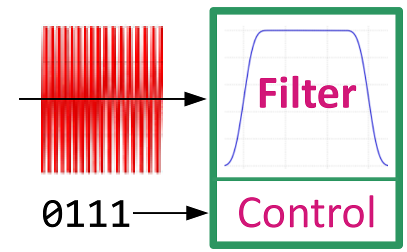
scientific analog

Writing UVM/SystemVerilog Testbenches for Analog/Mixed-Signal Verification

Scientific Analog, Inc.

21 June 2022


```
//Sequence code:
  start_item(TX_PKT);
  TX_PKT.randomize();
  finish_item(TX_PKT);
```



scientific analog

Contents

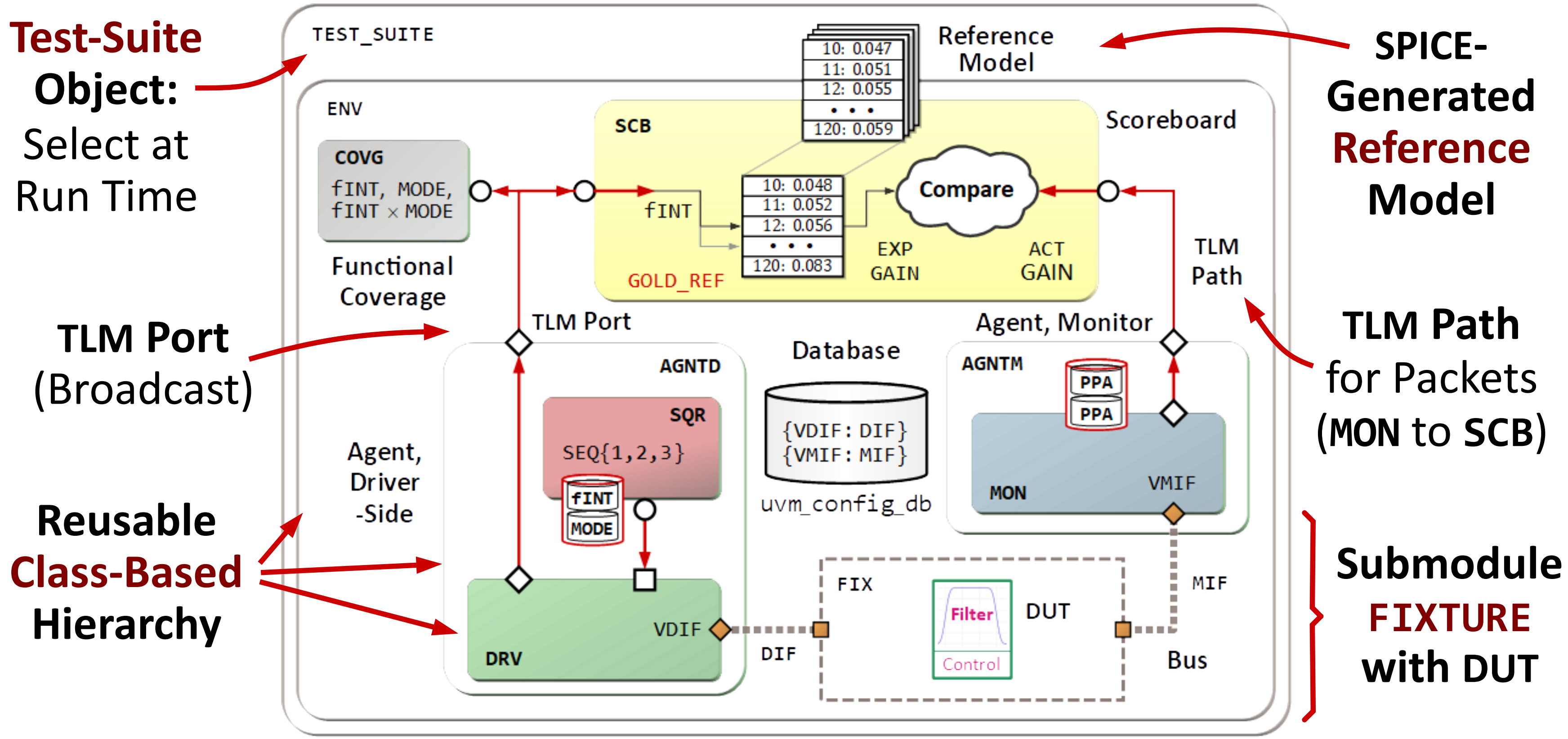
§1. Key UVM Features

- Organization of a UVM Testbench
- Classes Derive from a Base Library
- Packets Travel Over TLM Pathways
- Testbench Configured by Database
- Simulation is Run in Explicit Phases

Slide count: 7

Note:
Text in angle brackets (« ») indicates noncritical code, whose details are omitted to avoid cluttered slides.

UVM Testbench Organization



- Analog filter DUT instantiated in a fixture submodule.
- Virtual interfaces DIF, MIF connect it to the hierarchy.
- Packets travel between components via TLM paths.

Components from Base Class

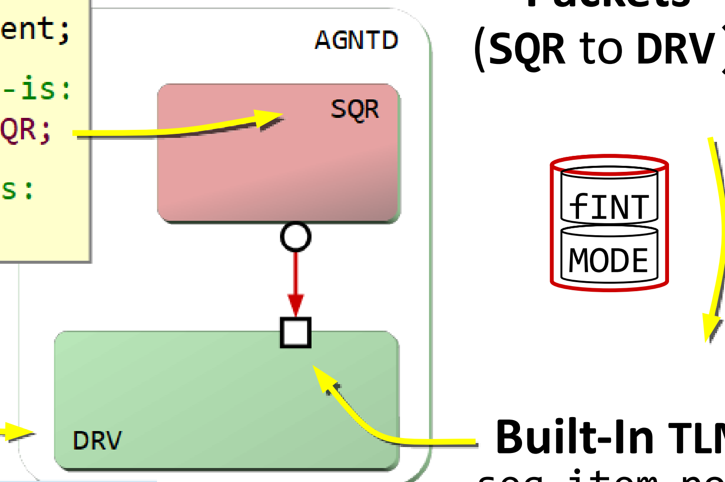
Typical Packaged Class Code

```
//Import base-class library:
  import uvm_pkg::*;
    .   .   .   .
//Derive agent from base class:
  class AGENTD extends uvm_agent;
    .   .   .   .
    //Instantiate base class as-is:
      uvm_sequencer #(PACKET) SQR;
    .   .   .   .
    //Instantiate extended class:
      DRIVER  DRV; .   .   .
```

Base Class Used As-Is

Driver-Side Agent

Packets (SQR to DRV)



Driver Extends a Base Class

```
//Derive DRIVER from base class:
  class DRIVER extends uvm_driver
    #(PACKET);  //Type parameter.
```

Built-In TLM seq_item_port

- UVM components (e.g. driver) derive from a base class.
- All members of `uvm_driver` are extended into `DRIVER`.
- Makes base-class methods and functionality available.

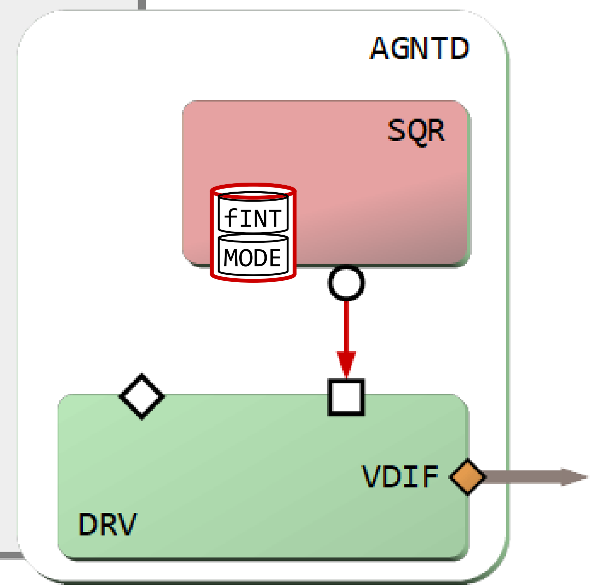
Objects from Base Class

Untimed Algorithm

```
class SEQ_FILTER extends uvm_sequence #(PACKET);
  PACKET  TX_PKT; //Declare packet.
  .   .   .   .   .
  task body(); //Generate packets.
    «Create new packet.»
    «Assign non-varying fields.»
    for («Loop for TRIALS»)
    begin:LOOP
      start_item(TX_PKT);  //Built-in.
      .   .   .   .   .
      «Update non-random fields. »
      «Randomize frequency, mode.»
      finish_item(TX_PKT); //Built-in.
    end:  LOOP
  endtask: body
```

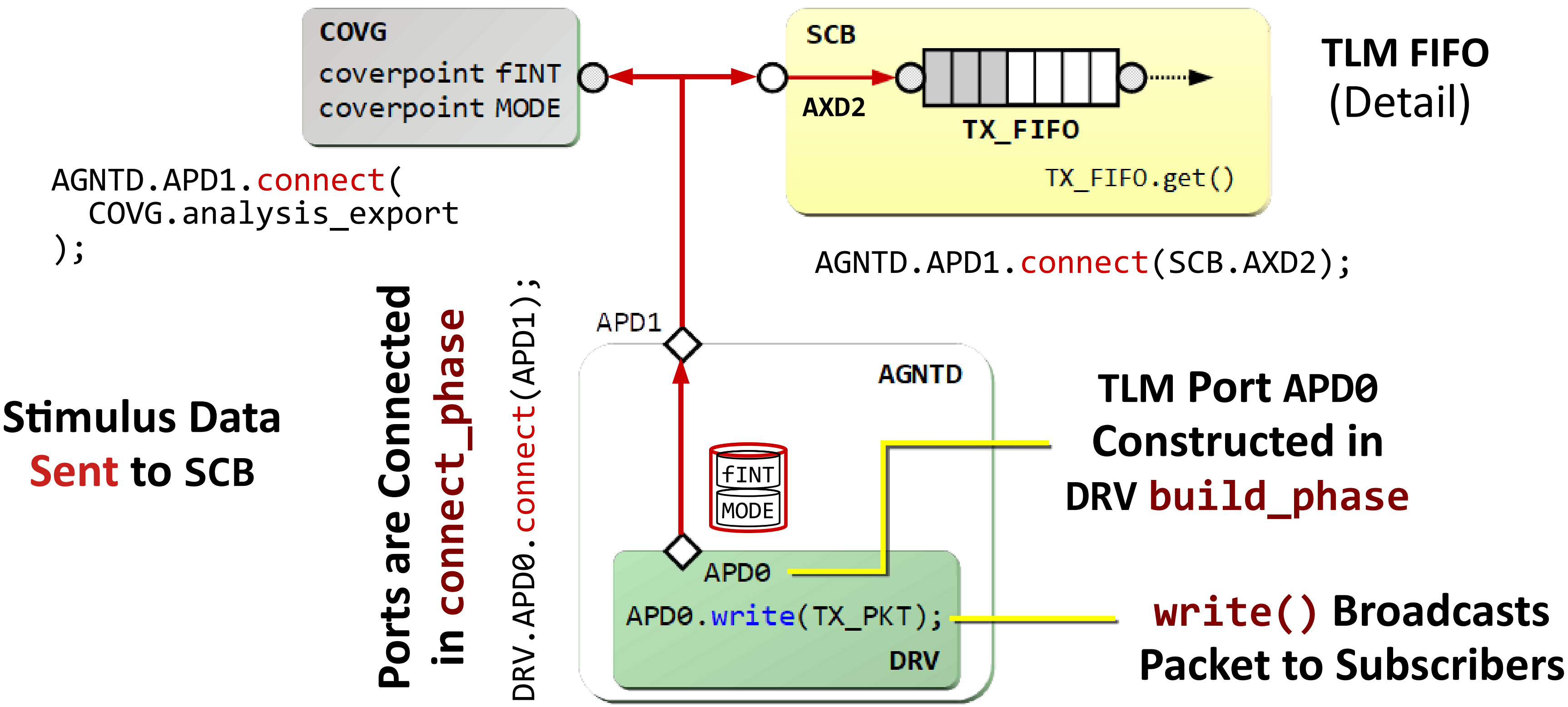
A Type Parameter

+TRIALS=3



- Not all UVM objects are structural parts of a testbench.
- Packets are transitory, yet still derive from a base class.
- Send each packet using built-in method `start_item()`.

Typical TLM Pathway



- OOP testbenches convey packets through the `mailbox`.
- UVM instead utilizes transaction-level TLM1 pathways.
- Each path is built by connecting its TLM ports together.

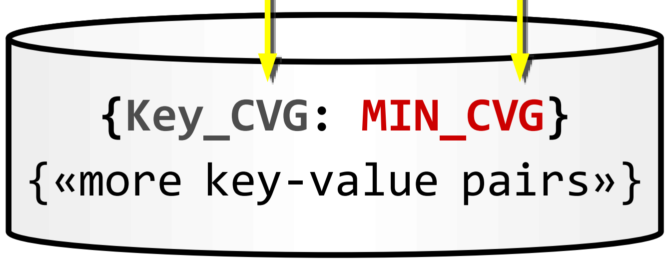
Configuring a UVM Testbench

```
COVG
coverpoint fINT
coverpoint MODE
```

Restricts `get()` to Instance COVG

```
//Knob MIN_CVG is defined in topmost testbench module:
  uvm_config_db #(real)::set(
    .cntxt(null), //Where in hierarchy to find value?
    .inst_name("uvm_test_top.E.COVG"),
  //DB resource: KEY           VALUE
    .field_name("Key_CVG"), .value(MIN_CVG)
  );
```

Methods **set/get()** Are Static

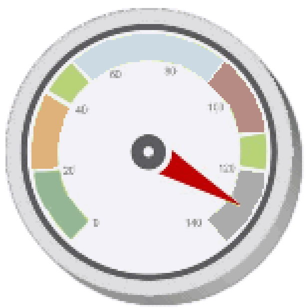


UVM Configuration Database

uvm_config_db is Detached from Hierarchy: Never Instantiated

- Configuration: a specific setting of knobs and switches.
- Externally adjust testbench's operation and topology.
- We'll use it to connect virtual to physical interface bus.

A Phased Simulation



Class uvm_phase

APD0 DRV

```
task run_phase(…);
  wait(!VDIF.RST);
  forever begin...
```

build_phase

connect_phase

. . . .

run_phase

extract_phase

. . . .

report_phase

. . . .

At Time 0.00:

- Component classes built.
- Configuration set/get().
- TLM ports interconnected.

Advance $time:

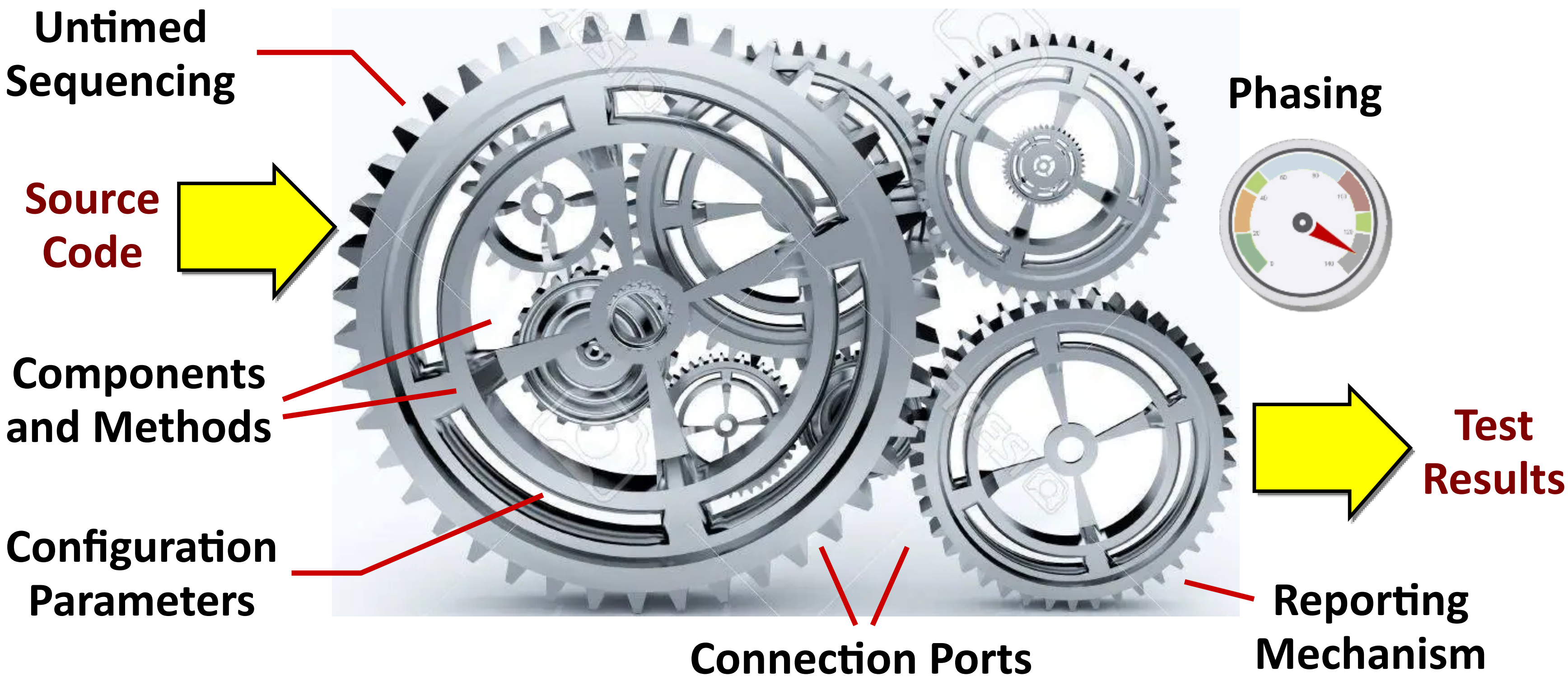
Run till all objections have been dropped, and UVM system calls $finish().

Finally:

- Extract scoreboard data.
- Check for discrepancies.
- Report test-suite results.

- UVM automates the successive stages in a simulation.
- Thus, in run phase, every run_phase task is executed.
- As phase begins, threads for these tasks are forked off.

A Configurable Machine



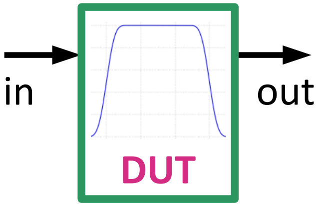
- Mechanical analogy: UVM like a configurable machine.
- Provides the infrastructure for any complex testbench.
- Utilize only the functionality you need to test the DUT.

scientific
analog

§2. From UVM to Analog

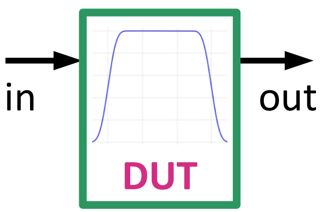
- Audio Bandpass-Filter DUT
- Auto-Extracted SystemVerilog
- A Fixture to Enclose the DUT
- Stimulus and Response Chains
- Applying Digital Control Bits
- Enumerated Filter Modes

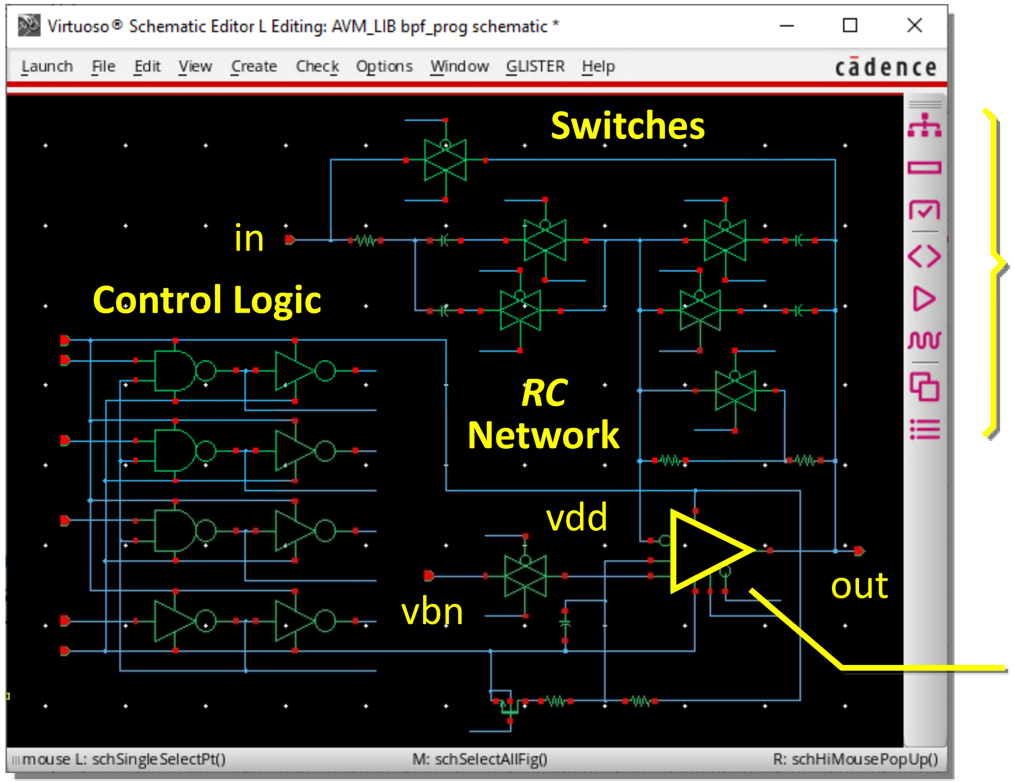
Slide count: 9



Audio Bandpass Filter

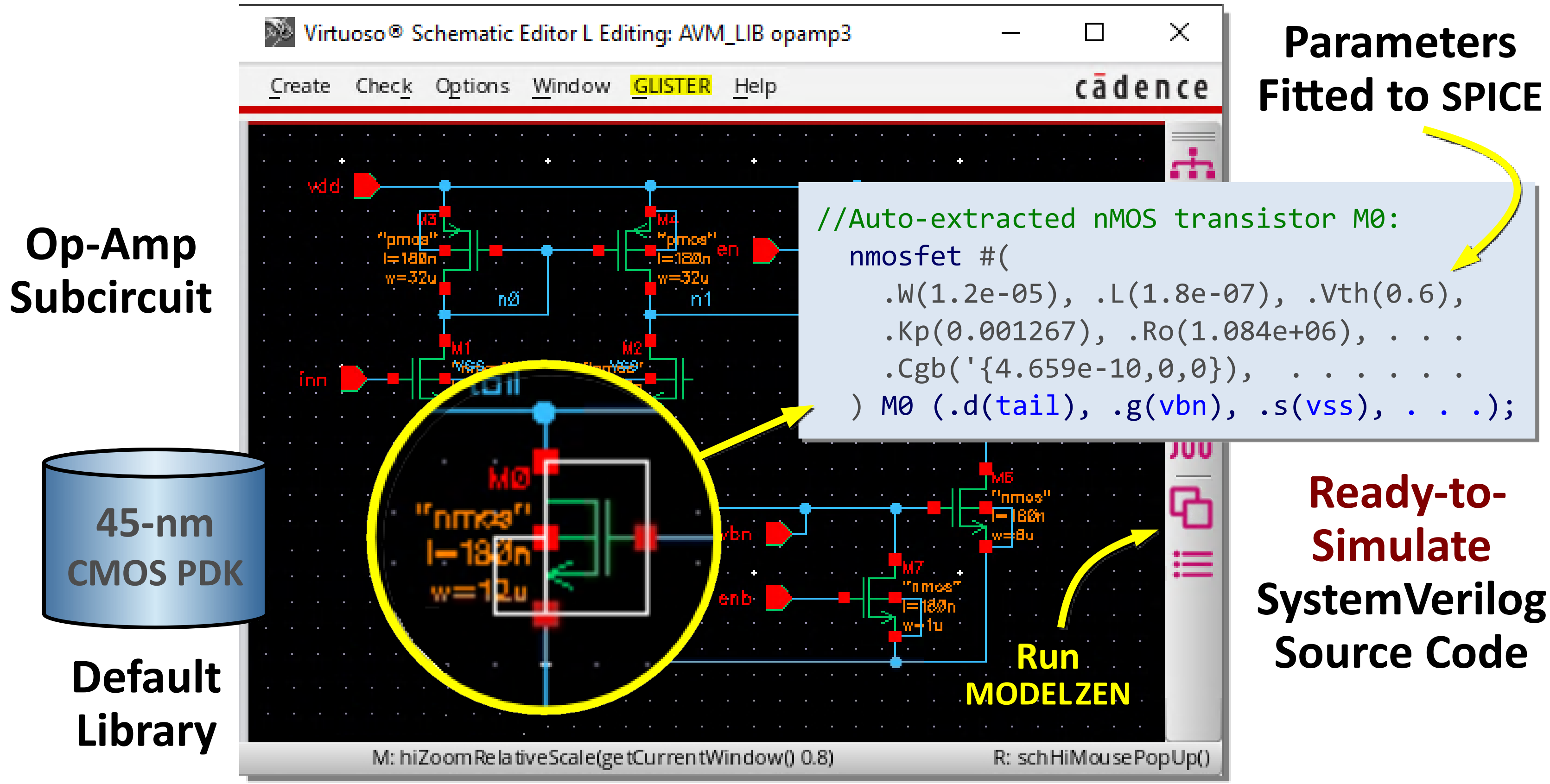
Programmable Automotive Bandpass Filter
(10–120 kHz)





- Active ***RC*** bandpass filter, with eight passband modes.
- Control logic switches ***R***, ***C*** elements in series or shunt.
- Designed in Virtuoso with transistors from 45-nm PDK.

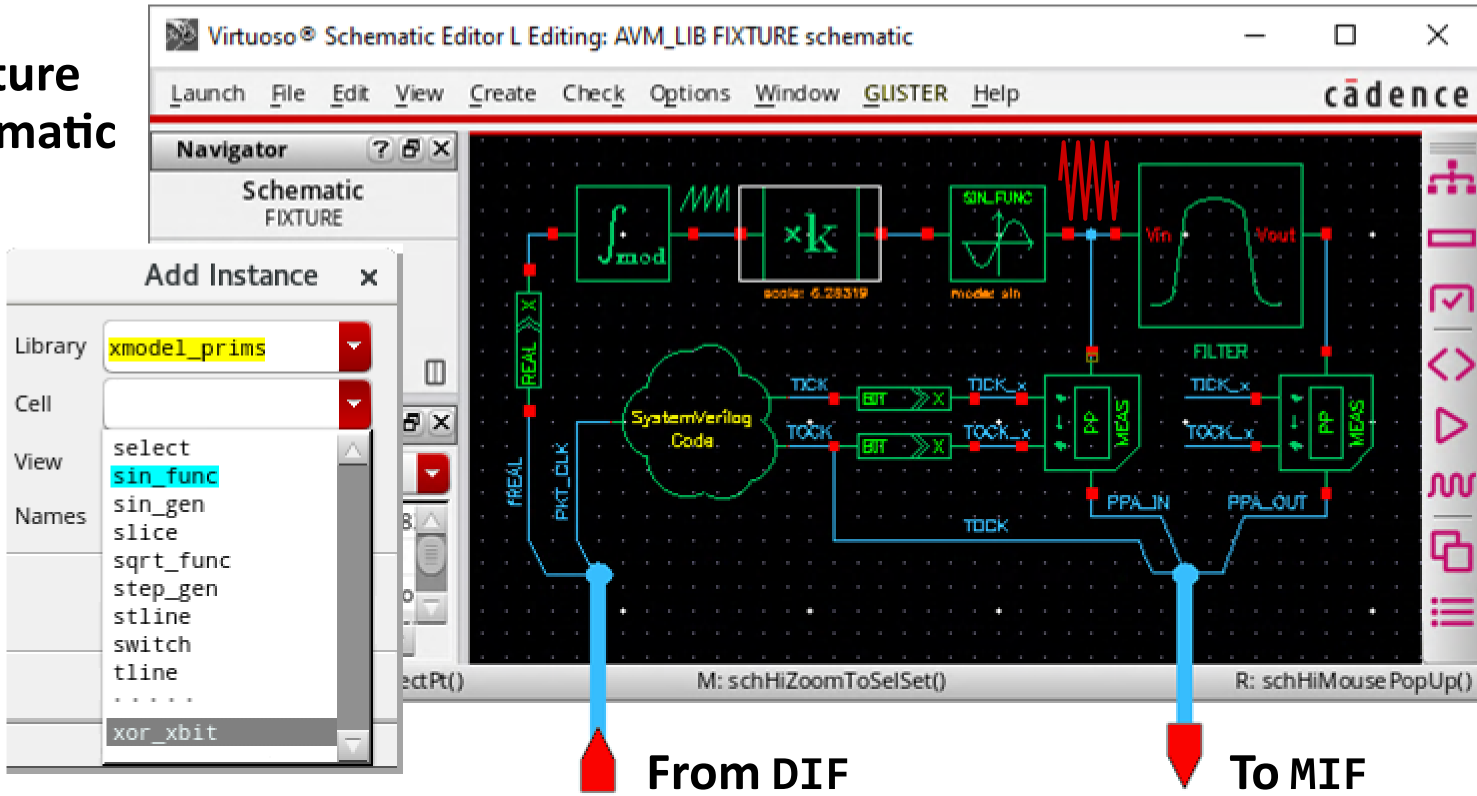
Auto-Extracted Models



- Extracted models are SPICE-accurate SystemVerilog.
- Bandpass filter's source code is thus sign-off quality.

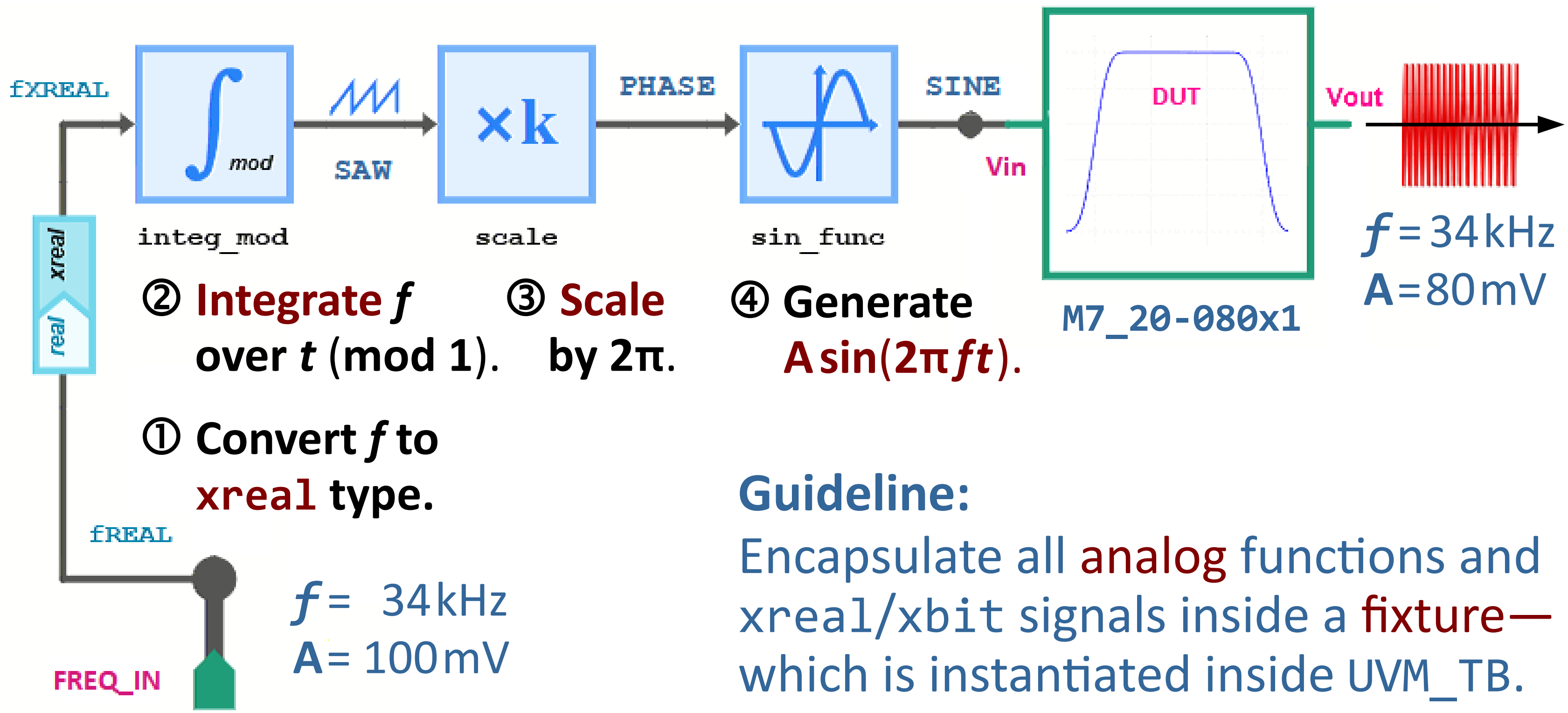
Sinusoidal Stimulus Chain

Fixture Schematic



- Module FIXTURE encloses DUT and instrumentation.
- This chain applies random-frequency sinusoid to DUT.
- Value *fREAL* is 10–120 kHz, arriving over interface DIF.

Example: Filter Stimulus



Guideline:
Encapsulate all analog functions and xreal/xbit signals inside a fixture—which is instantiated inside UVM_TB.

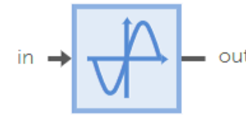
- Primitive ② integrates *f*, yielding a ramp waveform.
- Scaled by 2π, ramp becomes argument to sin($2\pi ft$).
- Generates a random-frequency sinusoidal DUT input.

An XMODEL Data Sheet

On-Line Data Sheet

PRIMITIVES / FUNCTIONS / SIN_FUNC

sin_func : A sinusoidal function for xreal.



The `sin_func` primitive produces an xreal-typed output which is a sinusoidal function of an xreal-typed input, where the function f() can be either sin() or cos() function.

```
out = f(in),
```

Terminal Names, Types

Input/Output Terminals

Name	I/O	Type
out	output	xreal
in	input	xreal

```
/* SystemVerilog instantiation
 * of primitive in the fixture:
 */
  sin_func
    #(.mode("sin"), .scale(VIN))
    F_SIN(.in(PHASE), .out(SINE)
  );
```

Parameters

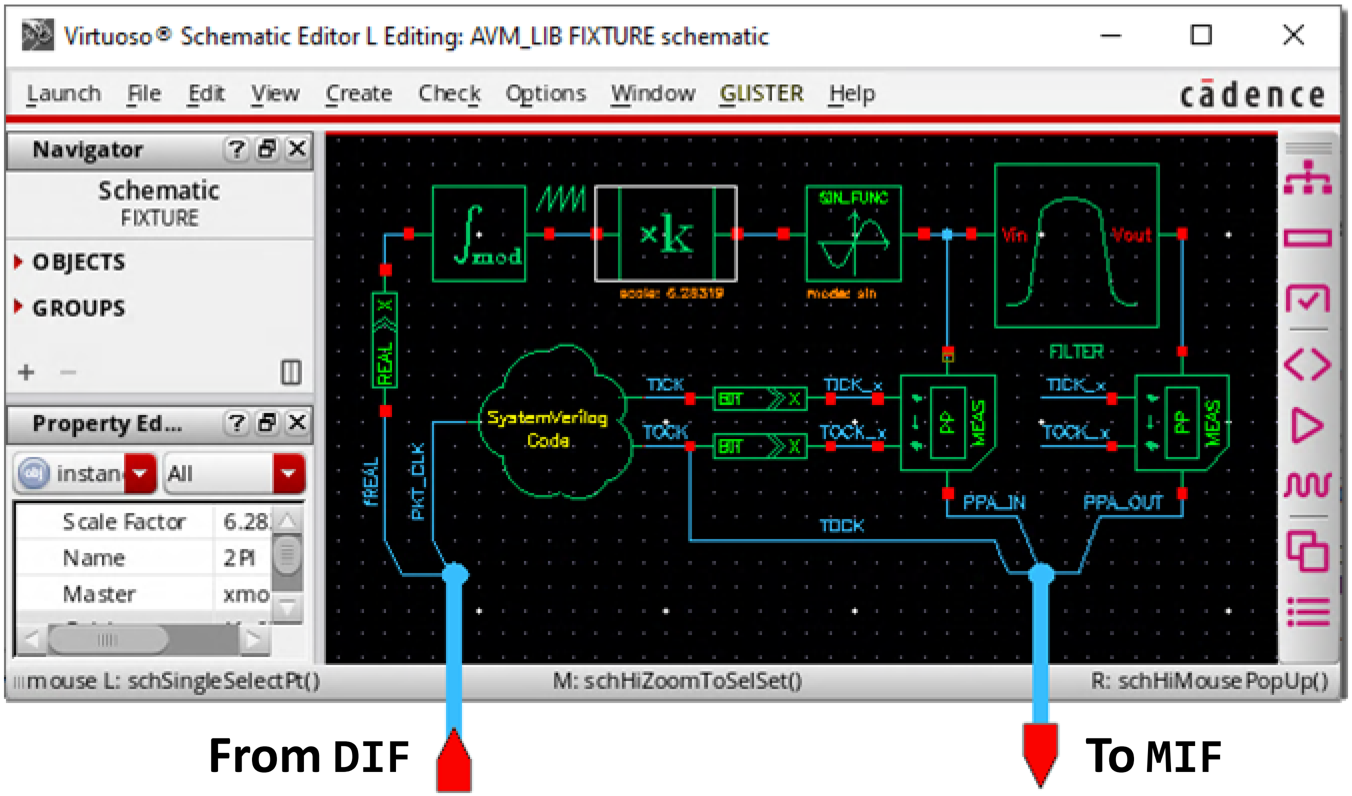
Name	Type	Default	Unit	Description
mode	string	'sin'	None	function type ("sin" or "cos")
scale	real	1.0	None	scale factor

Scaled by Parameter `VIN` (100 mV)

Parameter Names, Types

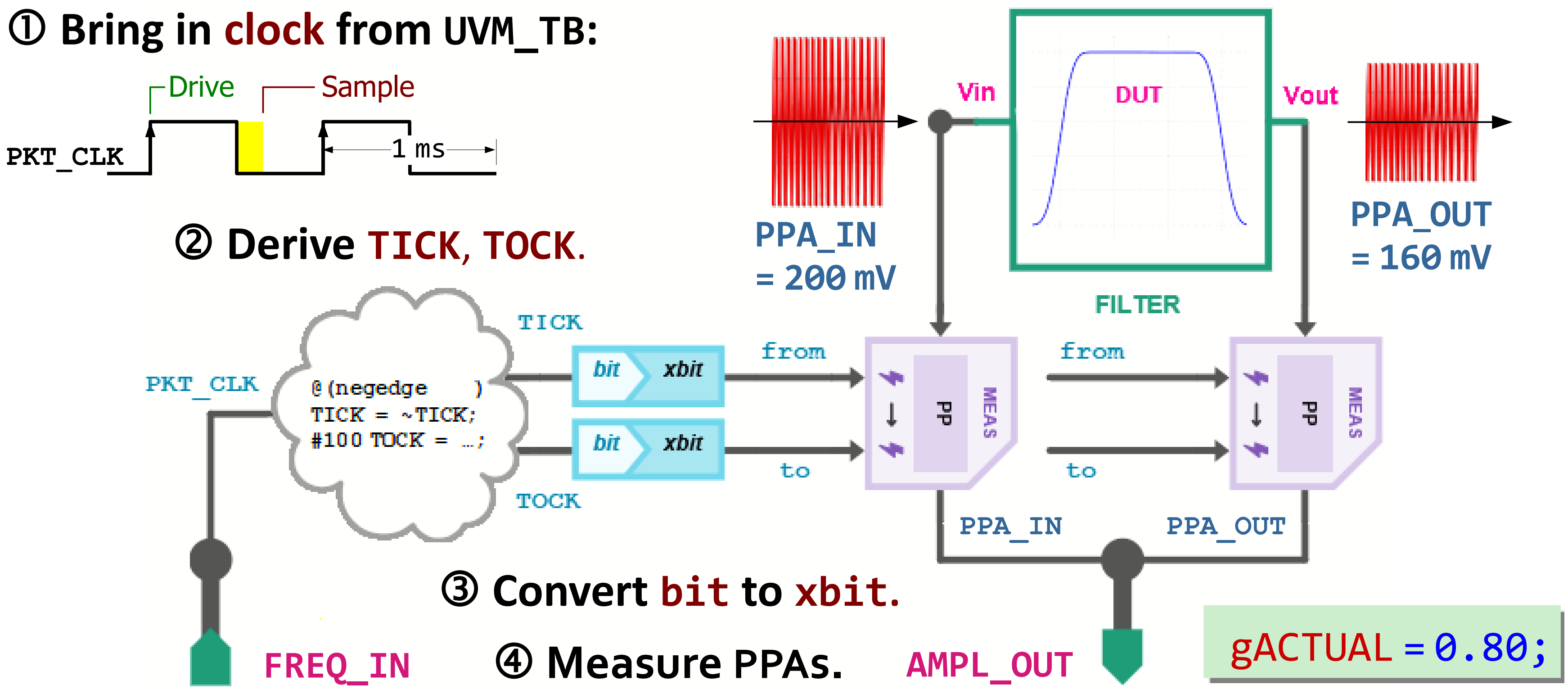
- Each primitive from XMODEL library has a data sheet.
- Place it in Virtuoso editor, or code it in SystemVerilog.
- Over a hundred primitives found in XMODEL library.

Sinusoidal Response Chain



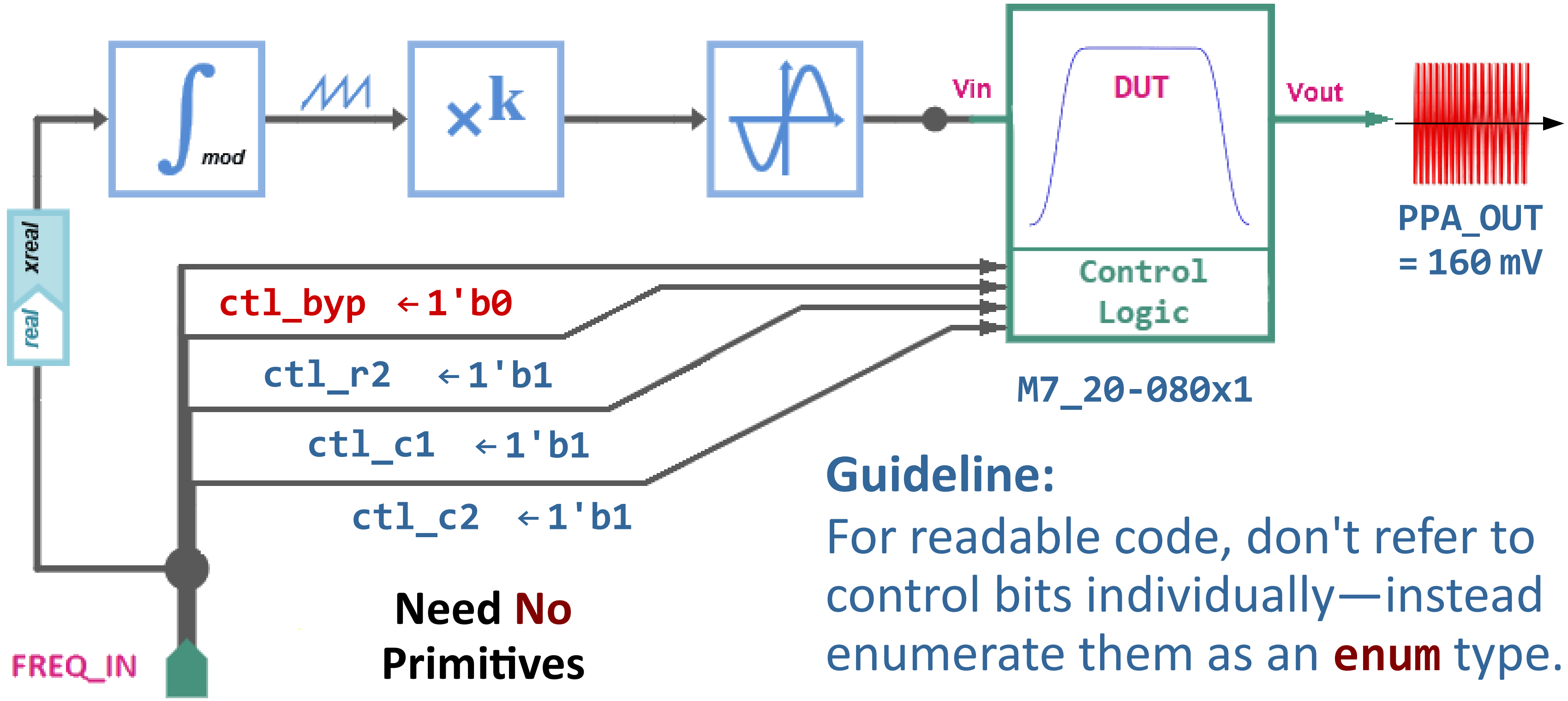
- Measure peak-to-peak amplitude at DUT input/output.
- Measurement time interval (100 µs) set by `TICK`, `TOCK`.
- Amplitudes then sent to monitor via interface bus `MIF`.

Example: Filter Response



- Primitives meas_pp require trigger signals: from, to.
- Derived from falling clock edge, to allow settling time.
- The outputs are real-valued peak-to-peak amplitudes.

Example: Digital Control Bits



Guideline:
For readable code, don't refer to control bits individually—instead enumerate them as an **enum** type.

- Control-logic input bits can be applied directly to DUT.
- Assert MSB to enter filter's bypass/power-down state.
- Lower three bits select among eight filtering modes.

DUT Digital Input Pins: {ctl_byp, ctl_r2, ctl_c1, ctl_c2}				
Binary Value	Enumeration Literal	f_{LO}	f_{HI}	$\lvert g \rvert$
0000	M0_40-060x2	40	60	2
0001	M1_40-040x2	40	40	2
0010	M2_20-060x2	20	60	2
0011	M3_20-040x2	20	40	2
0100	M4_40-120x1	40	120	1
0101	M5_40-080x1	40	80	1
0110	M6_20-120x1	20	120	1
0111	M7_20-080x1	20	80	1
1XXX	Bypass/Power-Down Mode			

```
//Enumerated filter modes:
  typedef
    enum bit [2:0] {
    //Symbolic Name     Code
      \M0_40-060x2 = 3'b000,
      \M1_40-040x2 = 3'b001,
      .    .    .    .    .    .    .
      \M6_20-120x1 = 3'b110,
      \M7_20-080x1 = 3'b111
    } MODE_t;
```

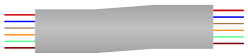
Escaped Names:
Leading Backslash, Trailing Space

- An **enum** type conveys a symbolic name and encoding.
- High-level UVM code can now refer to modes by name.
- Just package the **typedef**, and import where needed.

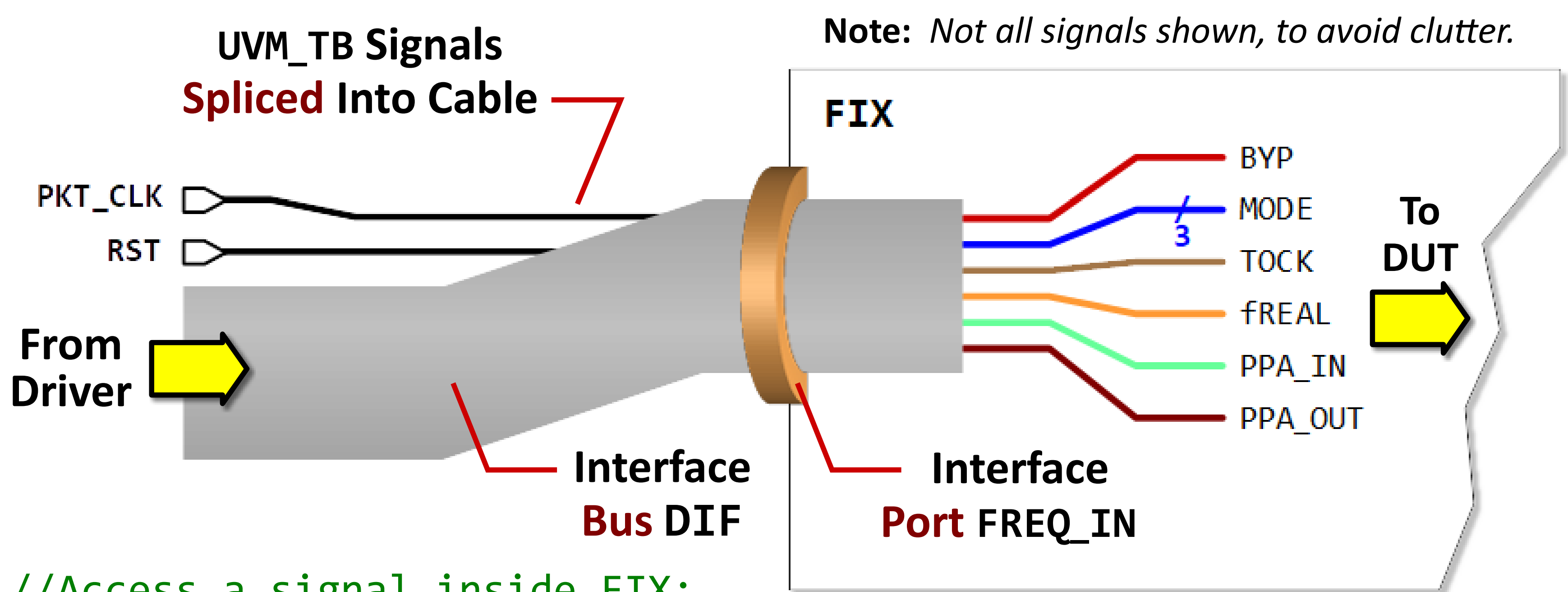
§3. Interface Bus Fabric

- Visualize an Interface Bus
- Describe an Interface Bus
- Physical Bus to Virtual
- UVM-AMS Compatibility
- Review Questions

Slide count: 6



Visualize an Interface Bus



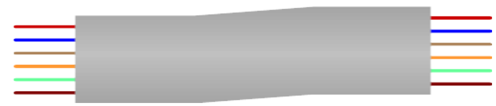

```
//Access a signal inside FIX:
  @(posedge FREQ_IN.PKT_CLK);
```

- Interface bus DIF brings all these signals into fixture.
- Signals can be of any variable or net type—even a UDT.
- Use dot notation to pick off any individual bus signal.

```
interface BAND_IF(«clock, reset»);
  import DATA_PKG::*;

//Control bits:
  bit BYP;
  MODE_t  MODE;

//Random frequency:
  int fINT; real fREAL;
   .   .   .   .   .   .
//Peak-peak amplitudes:
  real PPA_IN, PPA_OUT;
  bit TOCK; //Trigger.

endinterface: BAND_IF
```

Define Interface Buses

BAND_IF: DIF, MIF



```
module UVM_TB();
  import uvm_pkg::*;
  «Describe clock, reset.»

//Instantiate two buses:
  BAND_IF DIF(PKT_CLK, RST);
  BAND_IF MIF(PKT_CLK, RST);
   .   .   .   .
endmodule: UVM_TB
```

- An **`interface`** is a named bundle of related signals.
- The same description is reused for both `DIF` and `MIF`.
- Each utilizes only a subset of the signals in the bundle.

Physical Bus to Virtual (1/2)

Defined in DATA_PKG

```
//Virtual interface type:
  typedef  virtual BAND_IF  VIF_t;
```

Called From initial Block in Topmost Module

```
//UVM_TB module stores key-value pair:
  uvm_config_db #(VIF_t)::set(
    .cntxt(null), //Where in hierarchy?
    .inst_name("uvm_test_top.E.AGNTM.MON"),
  //DB resource: KEY              VALUE
    .field_name("Key_VMIF"), .value(MIF)
  );
```

Value Supplied Is Type VIF_t

Configuring Module Stores the Value
At Time 0.00

Configuration Database

- OOP testbenches utilize **new**() to connect VMIF to MIF.
- UVM stores physical bus instance MIF in its database.
- What is stored is actually a pointer to the bus instance.
- Value under name Key_VMIF can be retrieved by MON.

Physical Bus to Virtual (2/2)

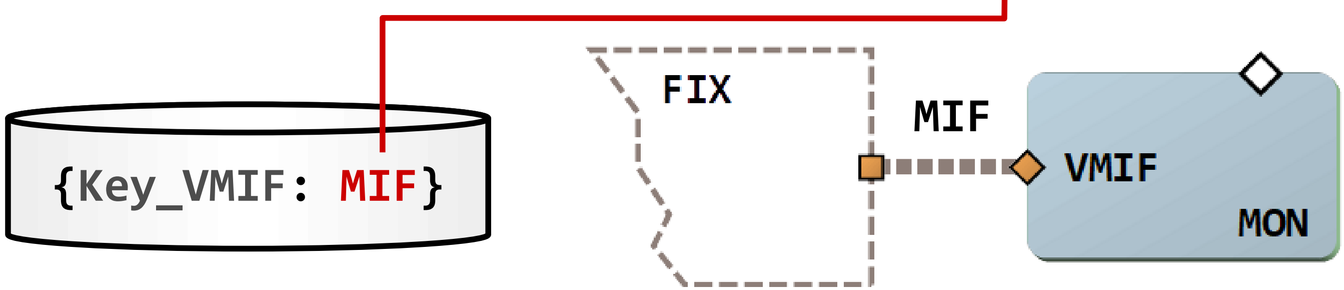
Declared in MONITOR

```
//Monitor's virtual-interface bus:
  VIF_t VMIF;
```

Called During build_phase of Monitor

```
  function void build_phase(uvm_phase phase);
    uvm_config_db #(VIF_t)::get(
      .cntxt(this),    //Monitor component itself.
      .inst_name(""), //Leaf cell with no children.
    //DB resource: KEY           VALUE
      .field_name("Key_VMIF"), .value(VMIF)
    );
```

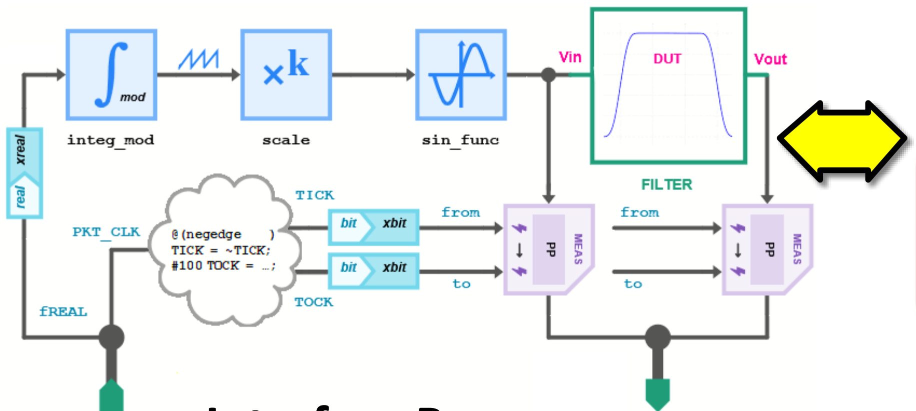
Configurable Component Retrieves Stored Value



- MON calls **get**() to retrieve value under the field name.
- Pointer to bus instance MIF known as virtual interface.
- VMIF is now a class variable pointing to a physical bus.

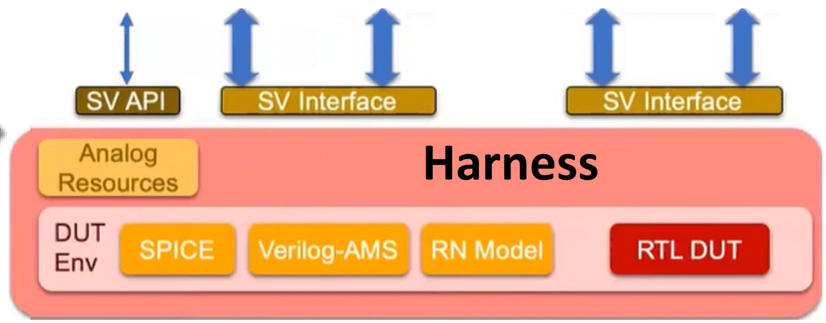
UVM-AMS Compatibility

Fixture



Interface Buses
to/from Class Hierarchy

Interface Buses
to/from Class Hierarchy



www.accellera.org/resources/videos/uvm-ams-workshop-2021

- XMODEL is compatible with new UVM-AMS standard.
- The above fixture is similar to the UVM-AMS harness.
- A key difference: no co-simulation facility is required.
- And the XMODEL library provides a ready-made API.

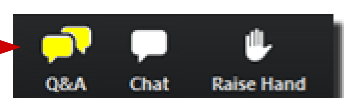
Matching Questions

1. A `class` cannot have any:
2. An `interface` bus must be:
3. Access uvm_config_db using:
4. Calling method `run_test()`:
5. A UVM testbench stops when:

a. static methods set, get().
b. all objections dropped.
c. I/O ports.
d. instantiated, to be useful.
e. will launch the test suite.

Answers:
1.c 2.d 3.a 4.e 5.b

Click to post question:

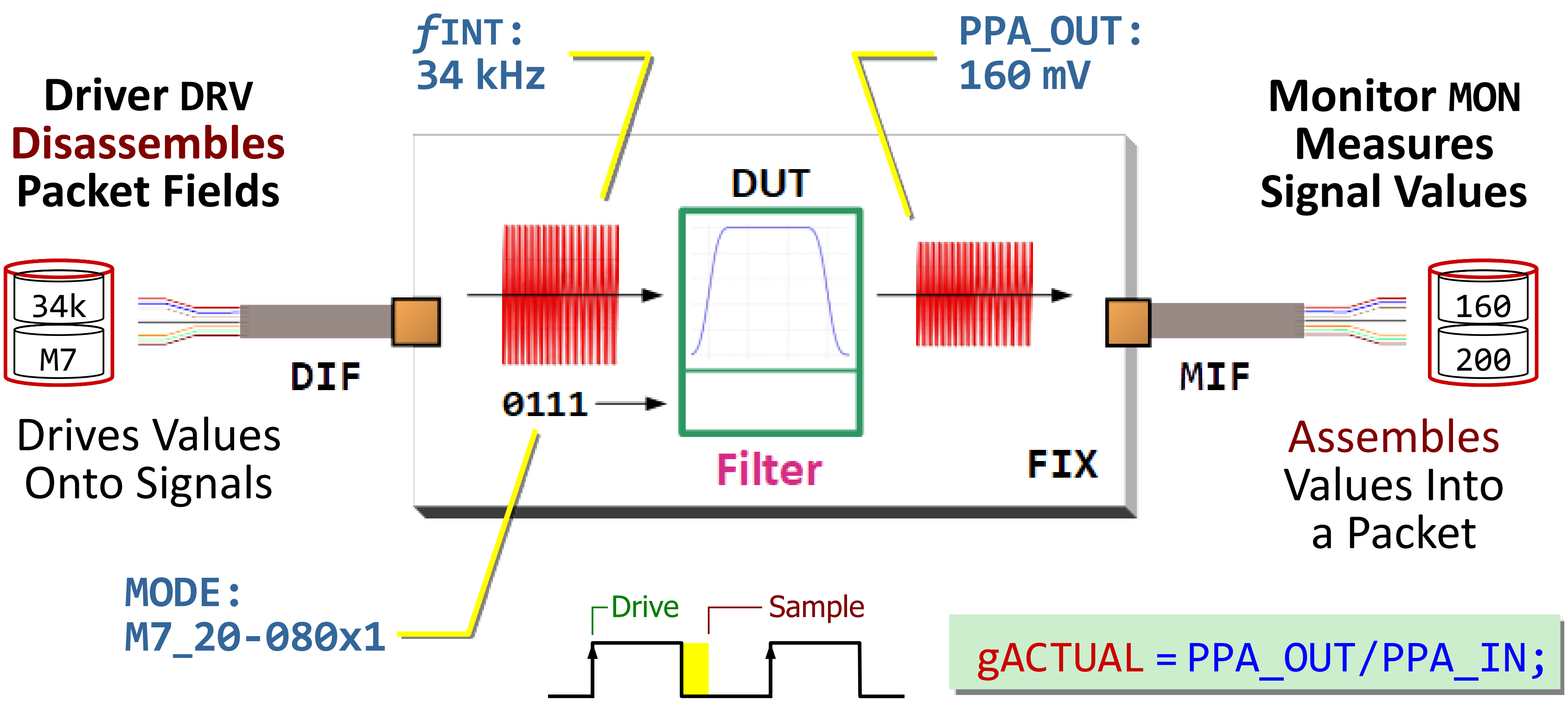


§4. A Basic Filter Test

- A Basic Test Scenario
- Why Use UVM Packets?
- Reusing the Packet Class
- Generate a Packet Stream
- Simulate the Packet Stream
- DUT-Specific Timing Budget

Slide count: 6

A Basic Test Scenario



- A basic test is to apply a random-frequency sinusoid.
- Put filter into one of eight modes, selected at random.
- Actual measured gain is ratio of the amplitudes, 0.80.

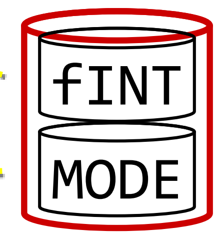
Why Use UVM Packets?

Typical UVM Packet

```
class PACKET extends uvm_sequence_item;
   .   .   .   .   .
//Stimulus fields:
   rand int fINT;     //10--120 kHz.
   real     fREAL;    //Equivalent.
   bit  BYP;          //Non-random.
   rand MODE_t MODE;  //M0..M7.
//Constrain fINT to design range:
   constraint fRANGE_con {
      fINT inside { [10:120] };
   }
   «continued on next slide»
```

Randomized Upon Calling .randomize()

Shaping Random Stimuli



Fields Applied to DUT Inputs

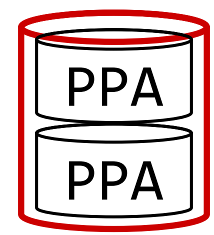
- Data in a packet class easily randomized, constrained.
- Using dynamic objects avoids out-of-memory issues.
- Packet class inherits key functionality from base class.
- Plain variables—array or **struct**—lack these features.

Reusing the Packet Class

If No Support for Random `real`

```
  «continued»
//Called after .randomize():
  function void post_randomize();
      .    .    .    .    .
  //Cast random fINT to a real:
    fREAL = real'(fINT * 1e3);
  endfunction: post_randomize

//Measured amplitude (volts):
  real PPA_OUT,
       PPA_IN;

endclass: PACKET
```

Measured Values from DUT Outputs



- Same packet used on monitor side, maximizing reuse.
- The monitor's fields of interest are: `PPA_OUT`, `PPA_IN`.
- Sent to scoreboard to compare with reference model.
- Not shown: queue, constraint to ensure cyclic modes.

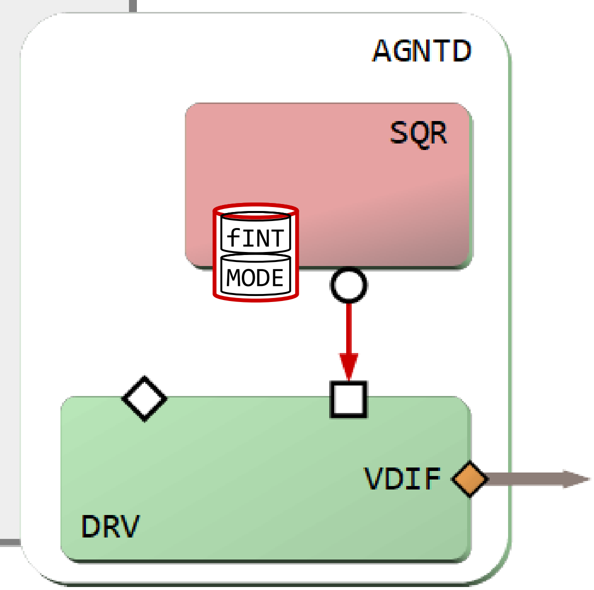
Generate a Packet Stream

Required Name

```
class SEQ_FILTER extends uvm_sequence #(PACKET);
  PACKET  TX_PKT; //Declare packet.
  .   .   .   .   .
  task body();  //Generate packets.
    TX_PKT = PACKET…create(…);
    TX_PKT.BYP = 1'b0; //Non-varying field.
    .   .   .   .   .
    for («Loop for TRIALS»)
    begin:LOOP
      start_item(TX_PKT);
      ++TX_PKT.TAG; //Integer ID field.
      TX_PKT.randomize();
      finish_item(TX_PKT);
    end:  LOOP
  endtask: body
endclass: SEQ_FILTER
```

Typical UVM Sequence

+TRIALS=3



- Sequence of untimed packets is sent out by sequencer.
- Driver applies fields to VDIF, till this sequence is done.

Simulate the Packet Stream

UVM Testbench: TRIALS=10

Cyclic Modes

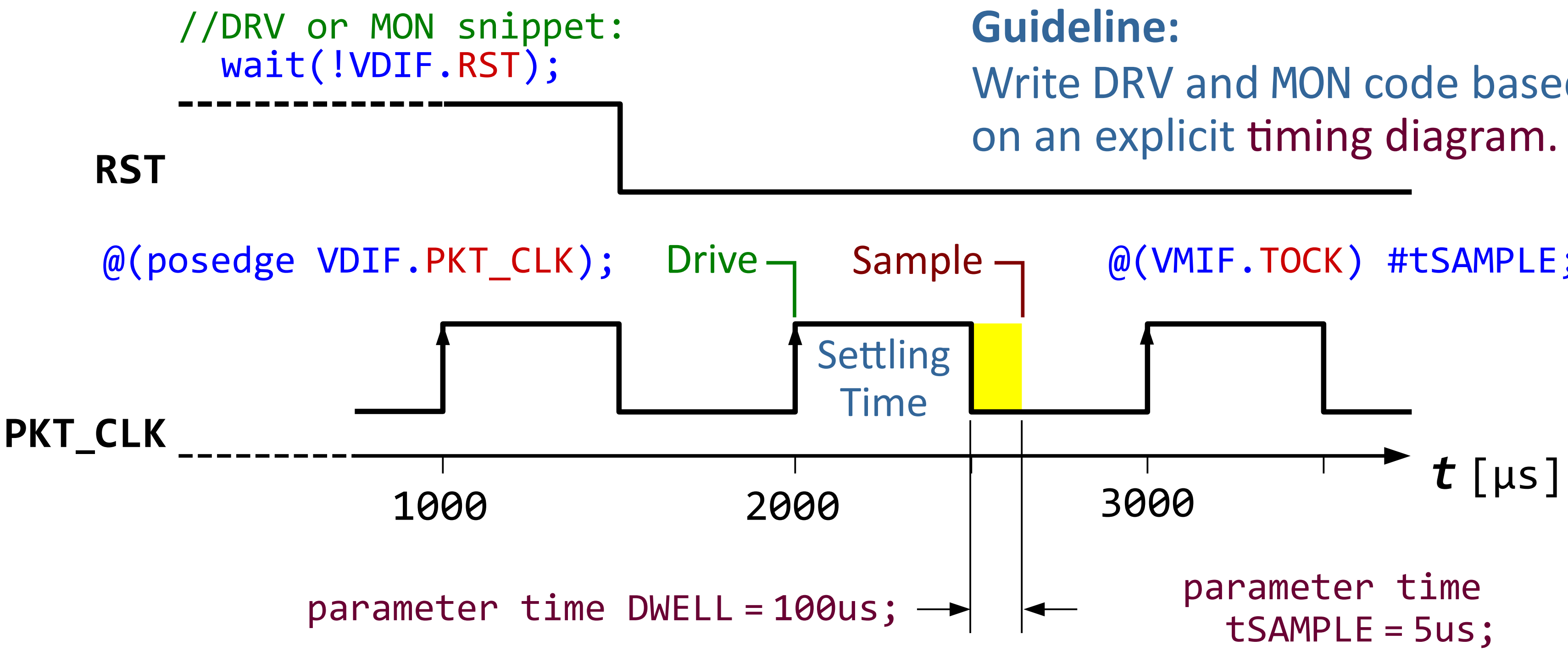
TX_ TAG	fINT kHz	MODE Name	STATE Name	RX_ TAG	gACT (OUT/IN)	gEXP (HSPICE)	gERROR (gACT-gEXP)
1	34	M7	FILTER	1	0.80081916	0.79615200	0.00466716
2	45	M0	FILTER	2	1.18788717	1.17873200	0.00915517
3	10	M4	FILTER	3	0.24548713	0.24592900	-0.00044187
4	16	M1	FILTER	4	0.68799809	0.68885000	-0.00085191
5	91	M5	FILTER	5	0.60613634	0.59471600	0.01142034
6	61	M2	FILTER	6	1.32845415	1.31148500	0.01696915
7	33	M3	FILTER	7	1.31424013	1.30375200	0.01048813
8	74	M6	FILTER	8	0.82776333	0.81830400	0.00945933
9	19	M7	FILTER	9	0.67933946	0.67720600	0.00213346
10	57	M2	FILTER	10	1.36394444	1.34703700	0.01690744

SPICE-Like Accuracy → Maximum |gERROR| per run: 0.01696915

Dave Rich, **randc Problem**, Verification Academy blog, 9 Jan 2013, verificationacademy.com/forums/systemverilog/randc-problem

- Modes are random-cyclic—repeat only after all used.
- When to apply mode and frequency is left up to driver.
- Similarly, when to measure PPAs is left to the monitor.

DUT-Specific Timing Budget



Guideline:
Write DRV and MON code based on an explicit timing diagram.

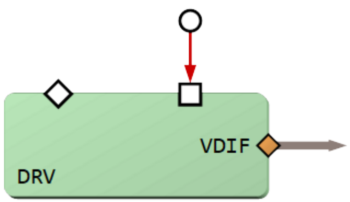
- UVM is unaware of your design-specific timing budget.
- We'll apply mode and frequency on active clock edges.
- And begin measuring PPAs upon inactive clock edges.
- Option: Use clocking block to specify all clock "skews."

scientific
analog

§5. Driver and Monitor

- Driver Setup Code
- Driver run_phase() Task
- The SEQ-DVR Handshake
- Monitor Setup Code
- Monitor run_phase() Task
- Building a Typical Agent

Slide count: 6



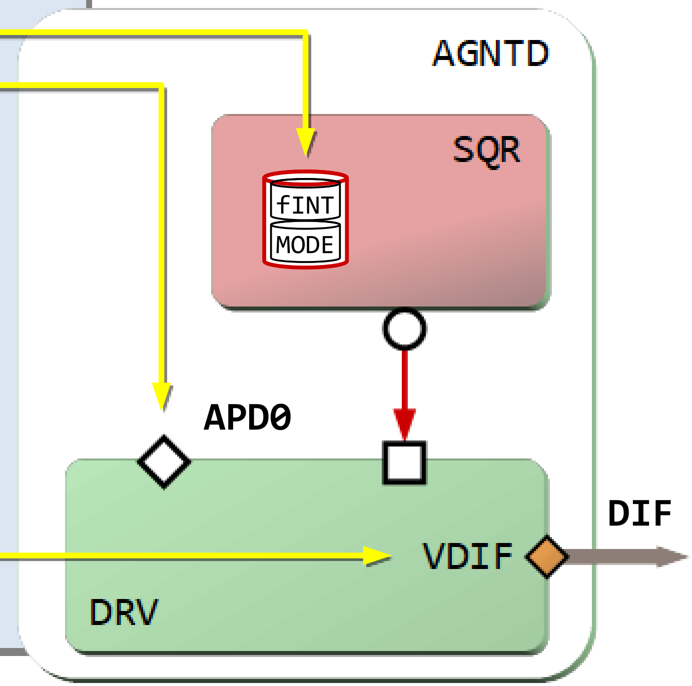
Driver Setup Code

Set-Up Code

Task Code

```
class DRIVER extends uvm_driver #(PACKET);
  VIF_t  VDIF;    //Virtual interface.
  PACKET TX_PKT; //Declare a packet.
  uvm_analysis_port #(PACKET) APD0;
  .   .   .   .   .
  function void build_phase(… phase);
    APD0 = new("APD0", this);
    uvm_config_db #(VIF_t)::get(…VDIF);
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    wait(!VDIF.RST);
    .   .   .   .   .
  endtask: run_phase

endclass: DRIVER
```

Driver Code

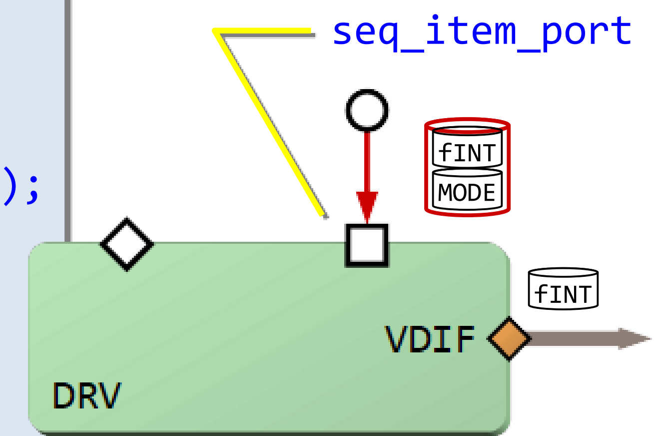


- Declares a packet—but does not need to create it.
- Constructs analysis port APD0—which is not built-in.
- Declares and gets a pointer VDIF to physical bus DIF.

Driver run_phase() Task


```
task run_phase(uvm_phase phase);
  wait(!VDIF.RST);
  forever
  begin:LOOP //Drive each active edge:
    @(posedge VDIF.PKT_CLK);
    seq_item_port.get_next_item(TX_PKT);
  //Bus Signal  Packet Field
    VDIF.BYP  = TX_PKT.BYP;
    VDIF.MODE = TX_PKT.MODE;
    VDIF.fINT = TX_PKT.fINT;
    .    .    .    .    .
    seq_item_port.item_done(); ...
  end:  LOOP
  .    .    .    .    .
endtask: run_phase
```

Built-In TLM Port
(From uvm_driver)



Sutherland & Fitzpatrick, **UVM Rapid Adoption**, DVCon 2015, §3.4.6 Driver/Sequence Synch.

- Each loop iteration begins at active edge of PKT_CLK.
- Calling get_next_item() pulls down packet from SQR.
- Disassemble the packet and assign each field to VDIF.

The SEQ-DRV Handshake

① **Sequence start_item() waits for driver to ask for the next packet.**

② **Driver get_next_item() pulls down item from SQR, waiting until its pointer is received.**

```
class SEQ_FILTER extends uvm_sequence...
  . . . . .
  task body();
    TX_PKT = PACKET…create(…);
    . . . . .
    for («Loop for TRIALS»)
    begin:LOOP
      start_item(TX_PKT);
      . . . . .
      TX_PKT.randomize();
      finish_item(TX_PKT);
    end:  LOOP
  endtask: body
endclass: SEQ_FILTER
```

Sequence

```
task run_phase(uvm_phase phase);
  wait(!VDIF.RST);
  forever
  begin:LOOP
    @(posedge VDIF.PKT_CLK);
    seq_item_port.get_next_item(…);
    VDIF.BYP  = TX_PKT.BYP;
    VDIF.MODE = TX_PKT.MODE;
    VDIF.fINT = TX_PKT.fINT;
    . . . . .
    seq_item_port.item_done();...
  end:  LOOP
  . . . . .
endtask: run_phase
```

Driver

Argument is TX_PKT

Stimulus Driven

③ **Sequence fills packet fields. Calling finish_item() sends completed packet to driver. Sequence loop then blocked.**

④ **Driver applies fields to VDIF. It calls item_done(), unblocking next iteration of the sequence.**

UVM carries out a complex sequence-driver handshake.

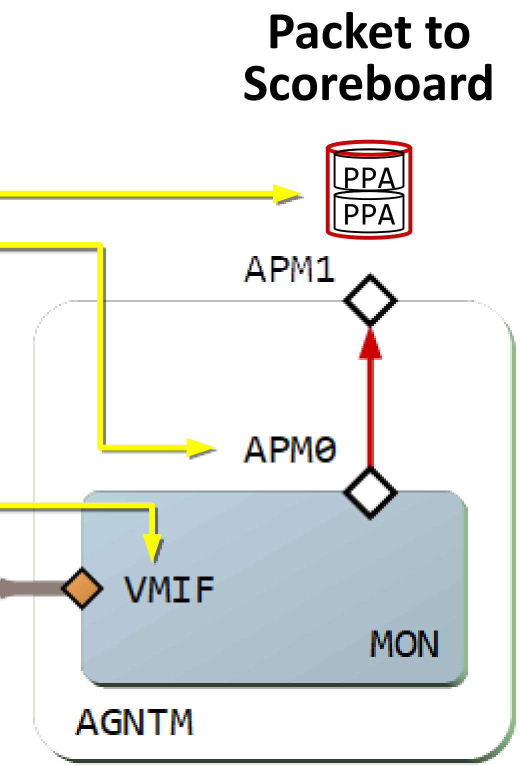
Monitor Setup Code

Set-Up Code

Task Code

```
class MONITOR extends uvm_monitor;
  parameter time tSAMPLE = 5us;
  VIF_t  VMIF;    //Virtual interface.
  PACKET RX_PKT; //Declare a packet.
  uvm_analysis_port #(PACKET) APM0;
  .  .  .  .  .
  function void build_phase(… phase);
    APM0 = new("APM0", this);
    uvm_config_db #(VIF_t)::get(…VMIF);
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    wait(!VMIF.RST);
    .  .  .  .  .
  endtask: run_phase
endclass: MONITOR
```

Monitor Code



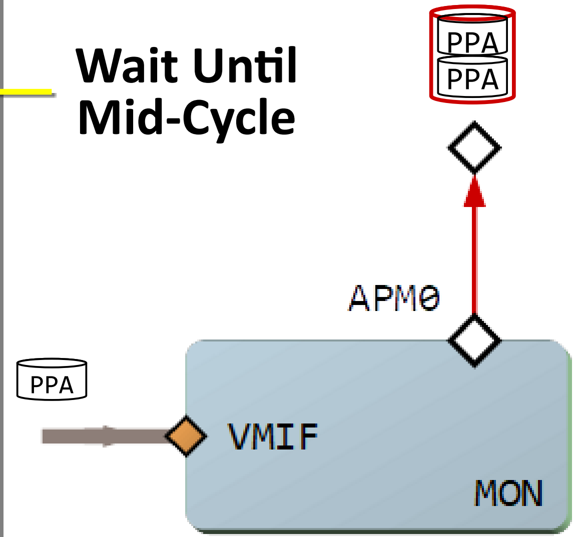
- Declares a packet—will later create it and fill its fields.
- Constructs analysis port APM0—which is not built-in.
- Declares and gets a pointer VMIF to physical bus MIF.

Monitor run_phase() Task


```
task run_phase(uvm_phase phase);
  wait(!VMIF.RST);
  forever
  begin:LOOP //Sample after a DWELL:
    @(VMIF.TOCK) #tSAMPLE;
    RX_PKT = PACKET…create("RX_PKT");
  //Packet Field     Bus Signal
    RX_PKT.TAG     = VMIF.TAG;
    RX_PKT.PPA_IN  = VMIF.PPA_IN;
    RX_PKT.PPA_OUT = VMIF.PPA_OUT;
    APM0.write(RX_PKT); //Send up.
    .   .   .   .   .
  end:  LOOP
endtask: run_phase
```

write() to Scoreboard

Wait Until Mid-Cycle



- Monitor assembles a packet from the signals on VMIF.
- Assigned about mid-cycle to appropriate packet field.
- Method write() sends up filled packet to scoreboard.

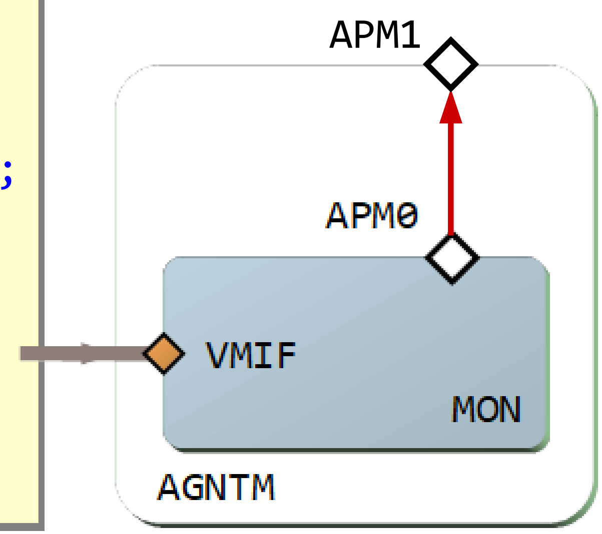
Building a Typical Agent

Construct

Factory-Create

```
//Monitor-Side Agent:
  class AGENTM extends uvm_agent;
    . . . . .
    uvm_analysis_port #(PACKET) APM1;
    MONITOR  MON;

    function void build_phase(…);
      APM1 = new("APM1", this);
      MON = MONITOR…create("MON", this);
    endfunction: build_phase

    function void connect_phase(…);
      . . . . .
      MON.APM0.connect(APM1);
    endfunction: connect_phase
  endclass: AGENTM
```

Guideline:
Ports are not factory-created—they never undergo a test-specific factory substitution.



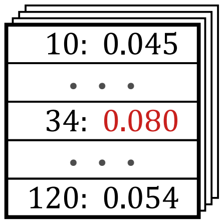
- Each agent handles a specific bus interface to the DUT.
- This agent creates the monitor MON and TLM port APM1.
- Connects existing MON port APM0 to its own port, APM1.

scientific analog

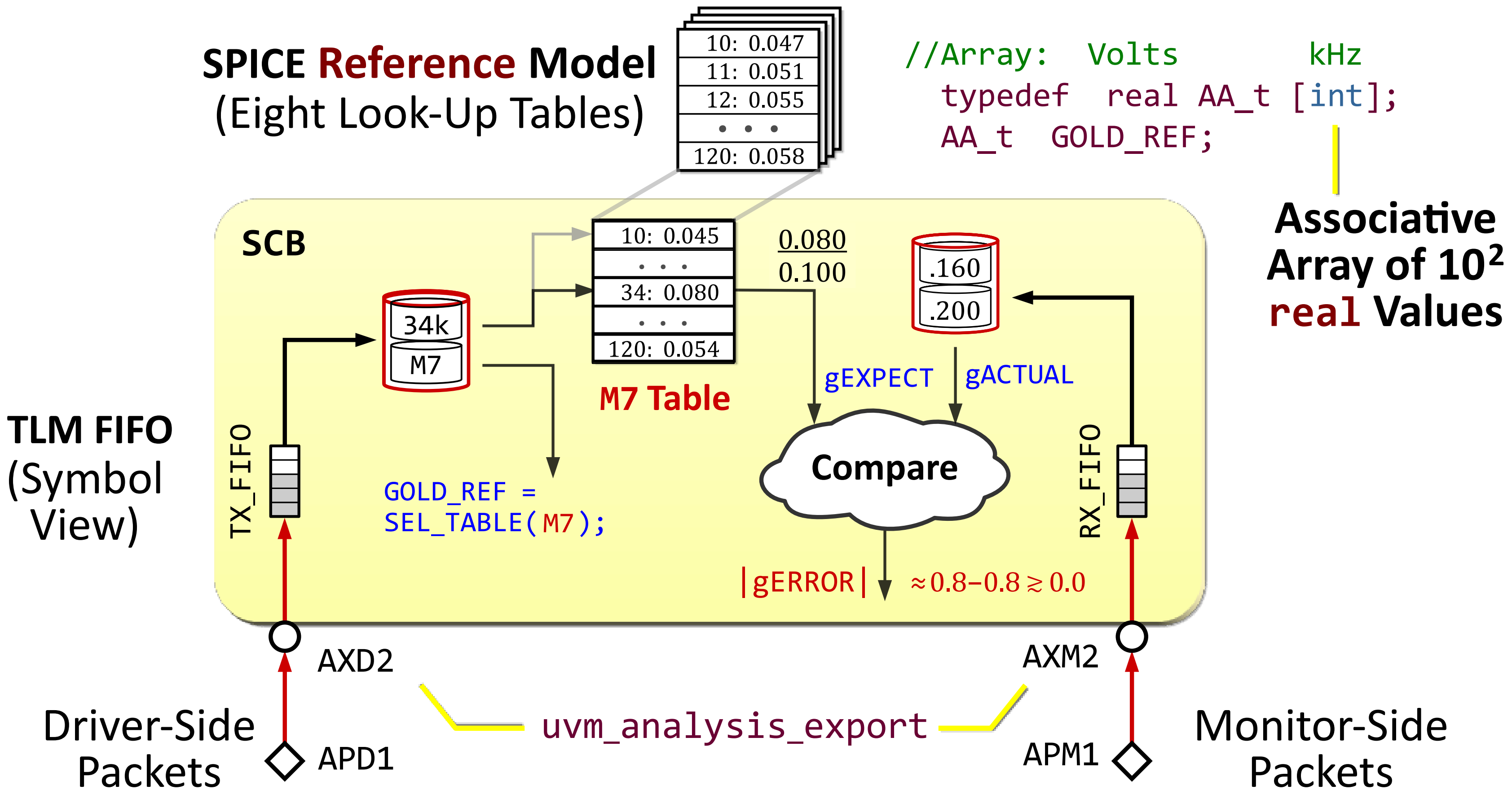
§6. A UVM Scoreboard

- Scoreboard Architecture
- Selecting the Right Table
- Scoreboard Setup Code
- SCB run_phase() Task
- SPICE to SystemVerilog
- A Scoreboard Transcript
- Review Questions

Slide count: 7



Scoreboard Architecture



- SCB compares actual measured gain against expected.
- Gets gEXPECT from SPICE-generated reference model.

Selecting the Right Table


```
//Array:  Volts      kHz
  typedef real AA_t [int];

//Unpacked array of tables:
  AA_t GOLD_REFS[8] = '{
    0: GOLD_REF_0,
    1: GOLD_REF_1,
    .  .  .  .  .
    7: GOLD_REF_7
  };
```

10:	0.047
11:	0.051
12:	0.055
• • •	
120:	0.058

Unpacked Array of Eight Tables

```
function AA_t SEL_TABLE(MODE);
  case (MODE) inside
    \M0_40-060x2 :
      return(GOLD_REFS[0]);
    \M1_40-040x2 :
      .  .  .  .  .
    \M7_20-080x1 :
      return(GOLD_REFS[7]);
    endcase
endfunction
```

10:	0.045
• • •	
34:	0.080
• • •	
120:	0.054

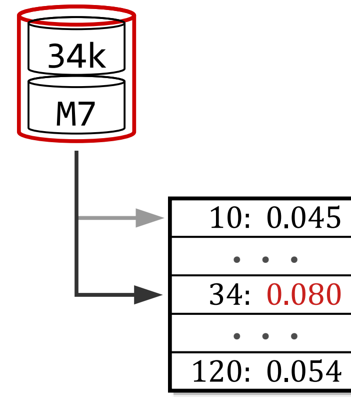
Look-Up Table for Mode M7

- Model comprises 8 look-up tables, one for each mode.
- The function SEL_TABLE() returns mode-specific table.
- Assigns it to a variable of type AA_t, named GOLD_REF.

Scoreboard Setup Code

Factory-Create

```
class SCOREBOARD extends uvm_scoreboard;
  .  .  .  .  .
  AA_t GOLD_REF; //Variable holding look-up table.
  real PPA_IN, PPA_OUT;  //Measured amplitudes.
  PACKET TX_PKT, RX_PKT; //DRV-side, MON-side.
  uvm_analysis_export   #(PACKET) AXD2, AXM2;
  uvm_tlm_analysis_fifo #(PACKET) TX_FIFO, RX_FIFO;
  .  .  .  .  .
  «function void build_phase()»
  «function void connect_phase()»
  «task run_phase()» //Scores only FILTER|RESUME.
  «function real GOLD_VOUT(int fINT_arg)»
  «function void extract_phase()»  //Max |gERROR|.
  .  .  .  .  .
endclass: SCOREBOARD
```



GOLD_VOUT(34 kHz)
→ 0.080 V

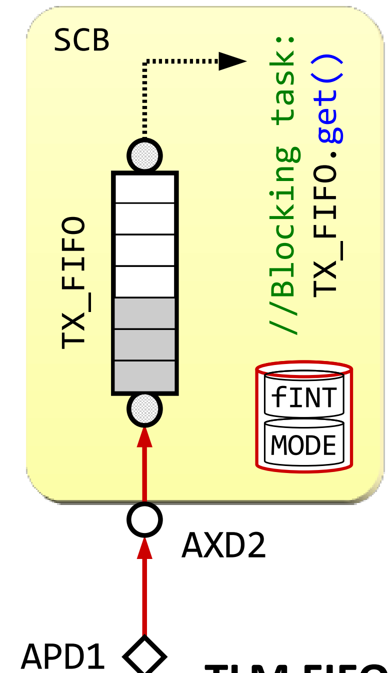
- Task run_phase() will compute gEXPECT and gACTUAL.
- Calls GOLD_VOUT() to look up VOUT expected for fINT.
- Next packets are retrieved from FIFOs by calling .get().

SCB run_phase() Task

Untimed Algorithm

```
task run_phase(uvm_phase phase);
  forever
  begin:SCORING //Loop till FIFOs empty.
    TX_FIFO.get(TX_PKT); //At 2.000 ms...
    RX_FIFO.get(RX_PKT); //At 2.605 ms...
    .   .   .   .   .
    GOLD_REF = SEL_TABLE(TX_PKT.MODE);
    .   .   .   .   .
    gACTUAL = RX_PKT.PPA_OUT/RX_PKT.PPA_IN;
  //From GOLD_REF look-up table:
    gEXPECT = GOLD_VOUT(TX_PKT.fINT)/VIN;
    gERROR  = gACTUAL - gEXPECT;
  end:  SCORING
endtask: run_phase
```

Cummings & Chambers,
UVM Analysis Port Functionality, Austin
SNUG-2018, §8. TLM FIFOs, p. 26



TLM FIFO
(Detail View)

- FIFOs allow packets with differing arrival times or rates.
- Use unbounded UVM FIFO: `uvm_tlm_analysis_fifo`.
- A `forever` loop keeps code independent of test length.

SPICE to SystemVerilog


```
.TITLE Bandpass Filter
****** ac analysis ***
freq       voltage m
  10.00k   45.0836m
  11.00k   48.4755m
  12.00k   51.6305m
  .  .  .    .  .  .
 117.00k   54.8377m
 118.00k   54.5270m
 119.00k   54.2188m
 120.00k   53.9132m
```

HSPICE Data

awk Script

```
//Associative array:
//Format: '{fINT: Vout, ...}
  real GOLD_REF_7[int] = '{
    int'( 10.00): 0.045_0836,
    int'( 11.00): 0.048_4755,
    int'( 12.00): 0.051_6305,
     .  .  .  .    .  .  .  .
    int'(119.00): 0.054_2188,
    int'(120.00): 0.053_9132,
         default: 0.000_0000
  };
```

SystemVerilog LRM, IEEE Std 1800-2017, **Clause 7.9.11**, Associative Array Literals

- Ran **HSPICE** linear AC analysis for frequency response.
- One `.lis` file is generated for each mode—a total of 8.
- Raw data converted by script to an associative array.

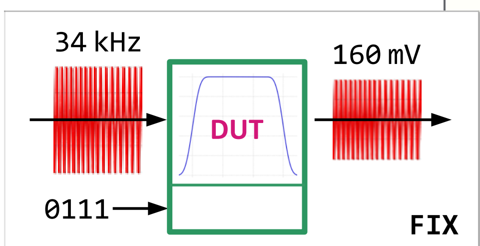
A Scoreboard Transcript

Transaction Details for UVM_HIGH

Tabular Scorecard Printout

UVM Testbench: TRIALS = 10

```
UVM_INFO @ 2.605 ms: uvm_test_top.E.SCB [SCORE]
  <TX(1)==RX(1)>
  In:   CTRL 4'b0_111, MODE M7, fINT  34 kHz
  Out: gACT = 0.16016383/0.200 = 0.80081916
  ----------------------------------------------------------------------
  TX_  fINT  MODE  STATE    RX_     gACT        gEXP         gERROR
  TAG  kHz   Name   Name    TAG   (OUT/IN)    (HSPICE)    (gACT-gEXP)
  ----------------------------------------------------------------------
    1   34    M7   FILTER     1  0.80081916  0.79615200   0.00466716
    2   45    M0   FILTER     2  1.18788717  1.17873200   0.00915517
    3   10    M4   FILTER     3  0.24548713  0.24592900  -0.00044187
    4   16    M1   FILTER     4  0.68799809  0.68885000  -0.00085191
    5   91    M5   FILTER     5  0.60613634  0.59471600   0.01142034
    6   61    M2   FILTER     6  1.32845415  1.31148500   0.01696915
    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
  ----------------------------------------------------------------------
                          Maximum |gERROR| per run:   0.01696915
```



- Scoreboard has SCORECARD sub-class for tabular print.
- Worst |gERROR| ÷ gEXPECT is only 1 or 2% of HSPICE.
- Next section shows how to run this test suite in UVM.

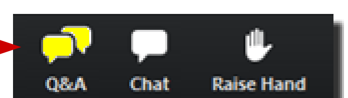
Multiple-Choice Questions

1. Each pair of packets (TX_PKT, RX_PKT) represents a single:
   a. sequence b. transaction c. phase.

2. Associative arrays allow look-up tables with an index that's:
   a. non-contiguous b. integer only c. enum only.

3. A cyclic random variable only starts repeating values when:
   a. simulation hangs b. seed is changed c. all values used.

Click to post question:

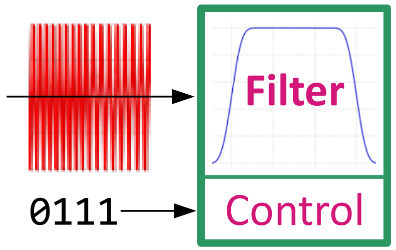


Answers:
1.b 2.a 3.c

§7. Running a Test Suite

- Topmost UVM Module
- A Test-Suite Component
- Simple Test-Suite Code
- UVM Testbench Topology
- A Typical Command Line

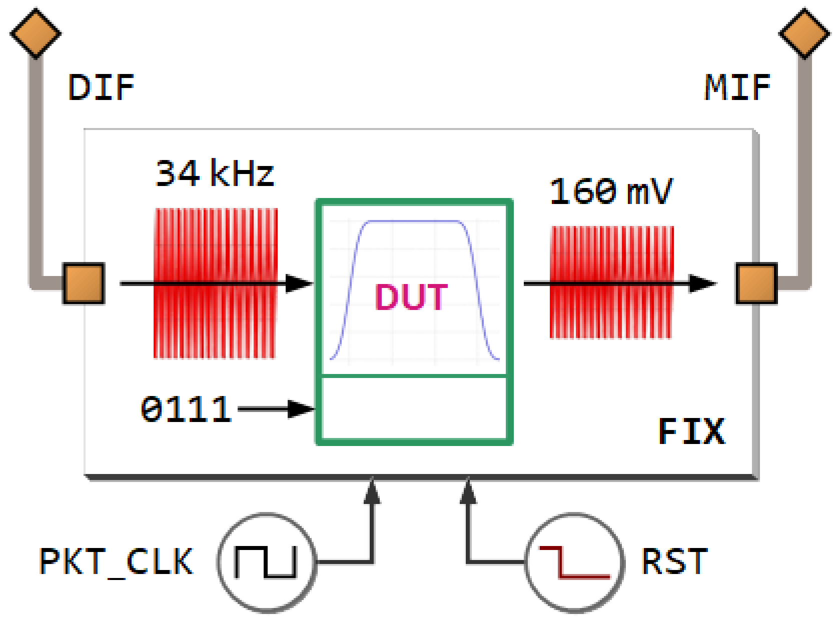
Slide count: 5



Topmost UVM Module

UVM_TB

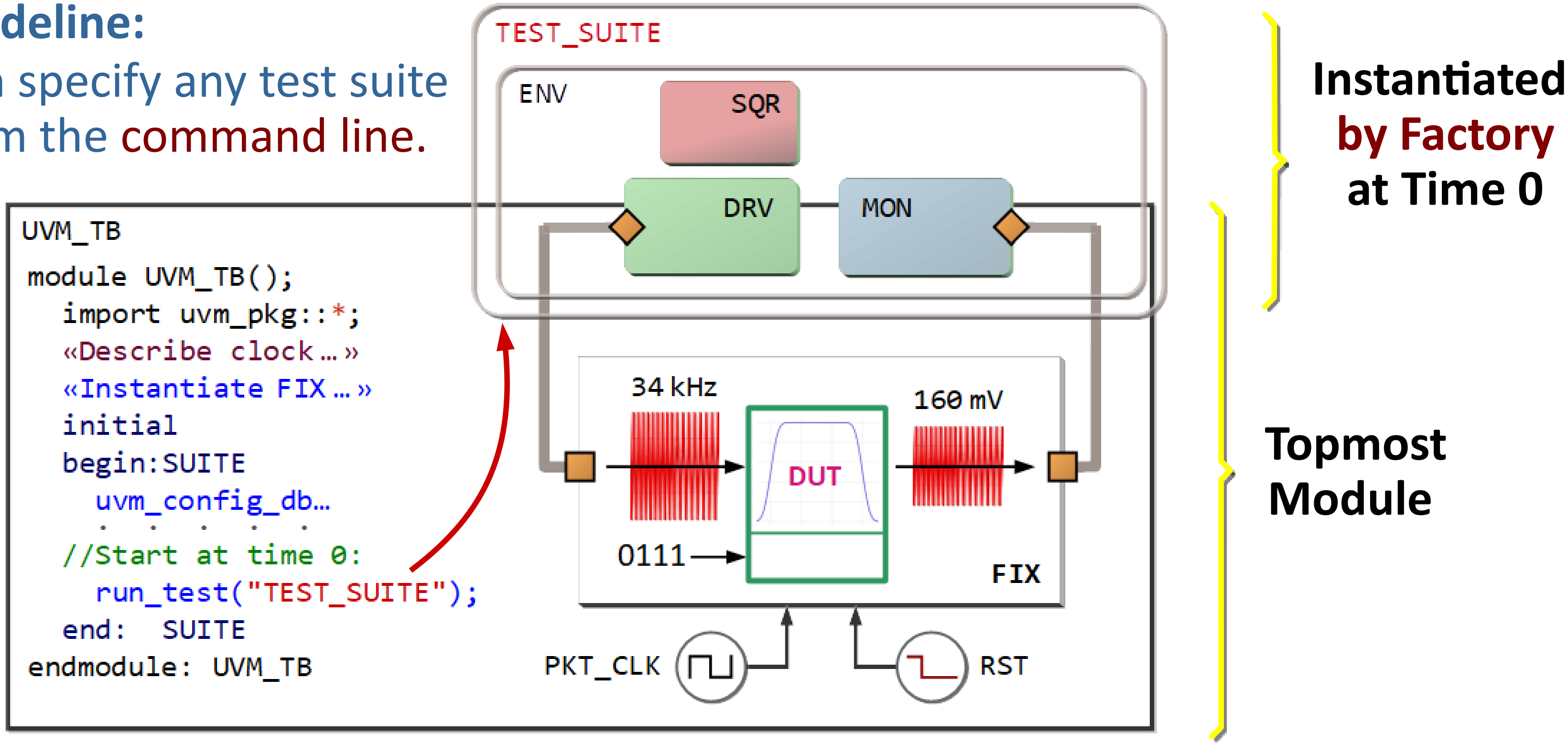
```
module UVM_TB();
  import uvm_pkg::*;
  «Describe clock, reset.»
  «Instantiate FIX, DIF, …»
  initial
  begin:SUITE
    uvm_config_db…::set(…);
    .   .   .   .   .


  end:  SUITE
endmodule: UVM_TB
```



- This module is the top block elaborated by a simulator.
- Instantiates fixture, DUT, and physical interface buses.
- Its `initial` block calls a uvm_root task: `run_test()`.

A Test-Suite Component

Guideline:
Can specify any test suite from the command line.



- Component TEST_SUITE never explicitly instantiated.
- Specify "TEST_SUITE" as the argument to run_test().
- Factory creates an instance of specified class at time 0.

Simple Test-Suite Code

Create Environment Top-Down

Launch SEQ1 on Sequencer

```
class TEST_SUITE extends uvm_test;
  .  .  .  .  .  .
  E = ENV::type_id::create("E",this);
  «Get knobs from command-line processor.»
  «After build_phase, print_topology().»
  .  .  .  .  .  .
  task run_phase(uvm_phase phase);
    SEQ_FILTER SEQ1; //Normal filtering.
    «scope».set_drain_time(this, 1000us);
    phase.raise_objection(this);
  //Launch sequence 1:
    SEQ1 = SEQ_FILTER::type_id::create("SEQ1");
    SEQ1.TRIALS = TRIALS; //Command-line knob.
    SEQ1.start(           ); //Run task body().
    phase.drop_objection(this);
  endtask: run_phase

endclass: TEST_SUITE
```

Allow Settling Time

Phasing System Halts Run

- Test suite in turn instantiates entire environment, E.
- Suite then launches a single sequence: SEQ_FILTER.

UVM Testbench Topology

UVM Testbench: TRIALS = 10

```
UVM_INFO @0.0 ms: [UVMTOP] UVM testbench topology
------------------------------------------------------------
Name                        Type                        Value
------------------------------------------------------------
uvm_test_top                TEST_SUITE                  @341
  E                         ENV                         @354
    AGNTD                   AGENTD                      @363
      DRV                   DRIVER                      @547
      SQR                   uvm_sequencer               @410
    AGNTM                   AGENTM                      @372
      MON                   MONITOR                     @602
    COVG                    COVERAGE                    @391
    SCB                     SCOREBOARD                  @381
      RX_FIFO               uvm_tlm_analysis_fifo #(T)  @711
      TX_FIFO               uvm_tlm_analysis_fifo #(T)  @652
------------------------------------------------------------
```

Class-Based Hierarchy

Not All Details Shown

Cummings & Fitzpatrick, **UVM Techniques for Terminating Tests**, DVCon 2011, §2. Compiling & Running UVM

- After ENV is built, test suite can call `print_topology()`.
- Factory created `TEST_SUITE` instance `uvm_test_top`.
- At time 0.0 ms, the hierarchy is ready for run_phase().

A Typical Command Line

Manifest File (Options; Source Files)

UVM Testbench: TRIALS = 10

```
62> cat man.f
--sim vcs
--top UVM_TB
--timescale 1us/1us
--elab-only
--elab-option -ntb_opts uvm-1.2 --
DATA_PKG.sv
GOLD_PKG.sv
SEQ_PKG.sv
BAND_IF.sv
DRV_PKG.sv
MON_PKG.sv
CVG_PKG.sv
SCD_PKG.sv
SCB_PKG.sv
ENV_PKG.sv
FILTER-zen.sv
FIXTURE.sv
UVM_TB.sv
62> xmodel -f man.f
62> simv  +TRIALS=10 +INACTV=5  +UVM_VERBOSITY=UVM_HIGH
62>
```

Topmost Module to be Elaborated

UVM Base-Class Library Version

Filter DUT with Auto-Extracted Models

Custom Knobs for Test Length

Supported Simulators:
- vcs
- xcelium
- ncverilog
- modelsim

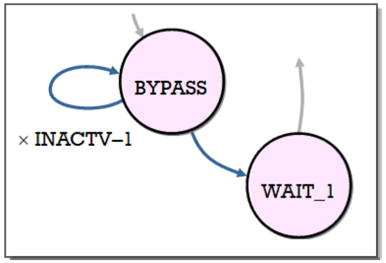
- Execute your simulator's command line using `xmodel`.
- This form compiles, elaborates source-code files listed.
- Then `simv` launches simulation, with run-time options.

scientific analog

§8. Extend the Test Suite

- Three Consecutive Sequences
- BYPASS Sequence Code
- Simulate the Extended Suite
- Employing Analog Assertions
- Test-Suite Waveforms

Slide count: 6



Three Consecutive Sequences

TRIALS →	INACTV →	fixed length →
SEQ1: SEQ_FILTER Perform basic filter test.	**SEQ2: SEQ_BYPASS** Power-down; bypass DUT.	**SEQ3: SEQ_RESUME** Resume basic filter test.

Extended Test-Suite Task

Repeat For Each Sequence

```
task run_phase(uvm_phase phase);
  SEQ_FILTER SEQ1; //Normal filtering.
  SEQ_BYPASS SEQ2; //Power-down state.
  SEQ_RESUME SEQ3; //Resume filtering.
   .   .   .   .   .
//Launch each sequence consecutively:



endtask: run_phase
```

Guideline:
UVM also supports:
- hierarchical;
- reactive;
- parallel sequences.

- Extend basic suite by powering down, then resuming.
- For each test, append another sequence to test suite.
- Adjust the sequence lengths using knobs like `TRIALS`.

BYPASS Sequence Code

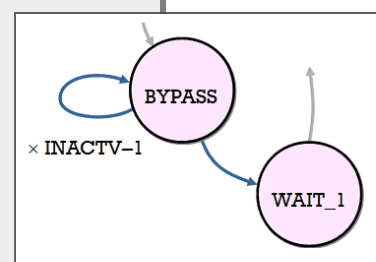
Task Code for SEQ2

These Packets Not Randomized

```
class SEQ_BYPASS extends uvm_sequence #(PACKET);
  PACKET  TX_PKT; //Declare packet.
  .   .   .   .   .
  task body();  //Generate packets.
    TX_PKT = PACKET…create("TX_PKT");
    .   .   .   .   .
    for (int J=1; J <= INACTV; J++)
    begin:LOOP
      start_item(TX_PKT); . . .
      TX_PKT.BYP = (J < INACTV)? 1'b1:1'b0;
      .   .   .   .   .
      finish_item(TX_PKT);
    end:  LOOP
  endtask: body
endclass: SEQ_BYPASS
```

WAIT_1 State

BYPASS (Power-Down) State



- Each sequence is a class with an untimed body() task.
- The **for** loop powers down filter for INACTV−1 cycles.
- Last iteration: single-cycle wait state to restore power.

Simulate Extended Suite

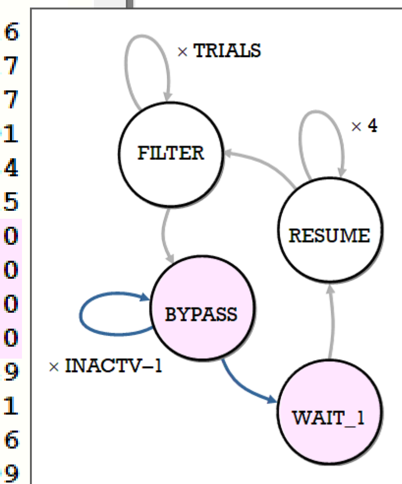
UVM Testbench: TRIALS = 6

TX_ TAG	fINT kHz	MODE Name	STATE Name	RX_ TAG	gACT (OUT/IN)	gEXP (HSPICE)	gERROR (gACT-gEXP)
1	34	M7	FILTER	1	0.80081916	0.79615200	0.00466716
2	45	M0	FILTER	2	1.18788717	1.17873200	0.00915517
3	10	M4	FILTER	3	0.24548713	0.24592900	-0.00044187
4	16	M1	FILTER	4	0.68799809	0.68885000	-0.00085191
5	91	M5	FILTER	5	0.60613634	0.59471600	0.01142034
6	61	M2	FILTER	6	1.32845415	1.31148500	0.01696915
7	XX	MX	BYPASS	7	0.00000000	0.00000000	0.00000000
8	XX	MX	BYPASS	8	0.00000000	0.00000000	0.00000000
9	XX	MX	BYPASS	9	0.00000000	0.00000000	0.00000000
10	XX	MX	WAIT_1	10	0.00000000	0.00000000	0.00000000
11	110	M3	RESUME	11	0.66836249	0.66027900	0.00808349
12	76	M6	RESUME	12	0.82291521	0.81331200	0.00960321
13	12	M4	RESUME	13	0.29037514	0.29073300	-0.00035786
14	63	M6	RESUME	14	0.85088399	0.84234200	0.00854199

SEQ1 (rows 1–6), SEQ2 (rows 7–10), SEQ3 (rows 11–14)

SPICE Accuracy → Maximum |gERROR| per run: 0.01696915

Four DUT States

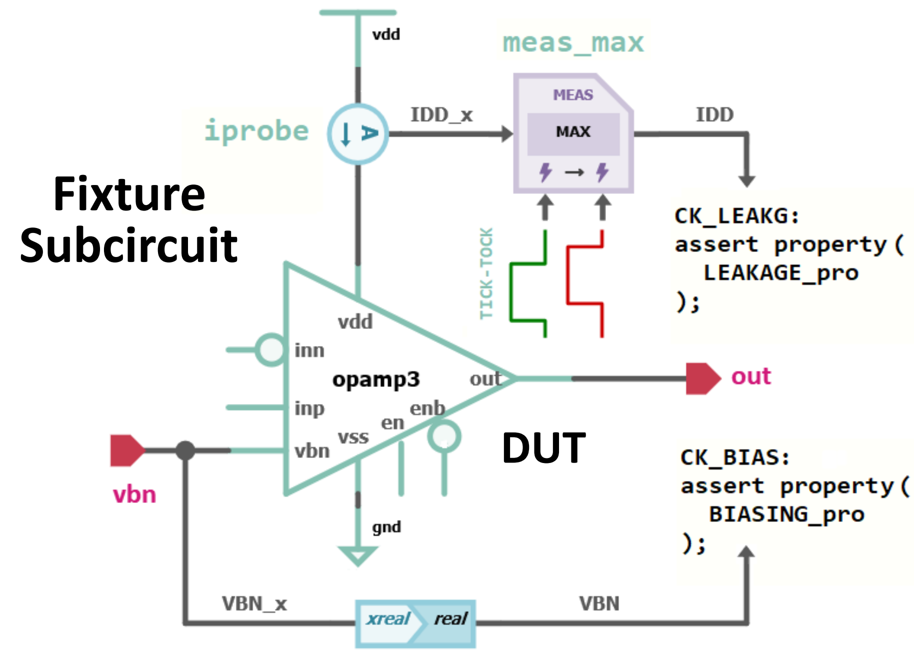


- The sequences run consecutively, with no gaps in time.
- An optional STATE variable aids in monitoring progress.
- State RESUME tests a recovery from power-down mode.

Using Analog Assertions

Analog/Mixed-Signal Assertions for I, V		
Assertion Label	CK_LEAKG	CK_BIAS
Variable Checked	Idd (Leakage)	Vbn (nMOS Bias)
Antecedent Condition	STATE == BYPASS	STATE == WAIT_1
Property Expression	Idd(max) < 10 nA	Vbn = 700 ± 50 mV

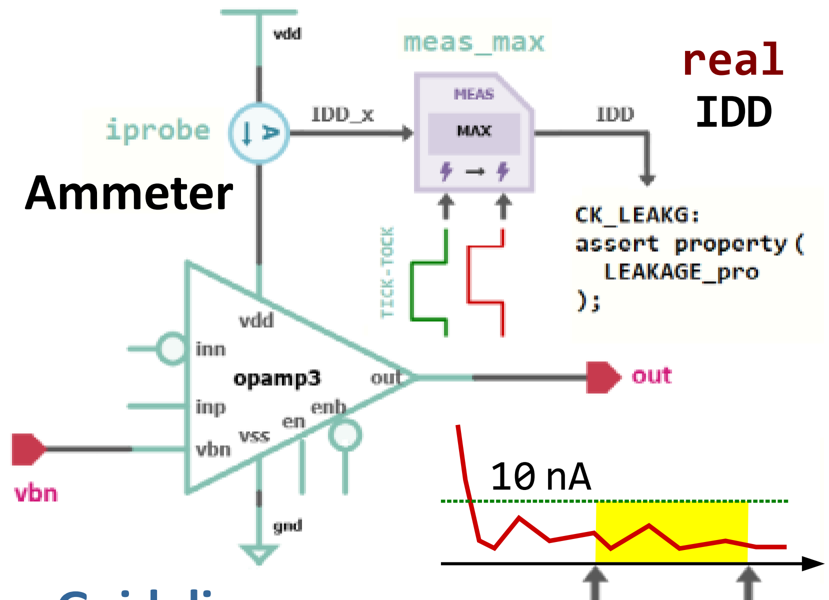
Assertion Plan



- Filter properties can be verified even in inactive cycles.
- Analog assertions check leakage current and bias level.
- Instrumented by connecting a few XMODEL primitives.
- Assertions complement self-checking UVM testbench.

Coding an Analog Assertion

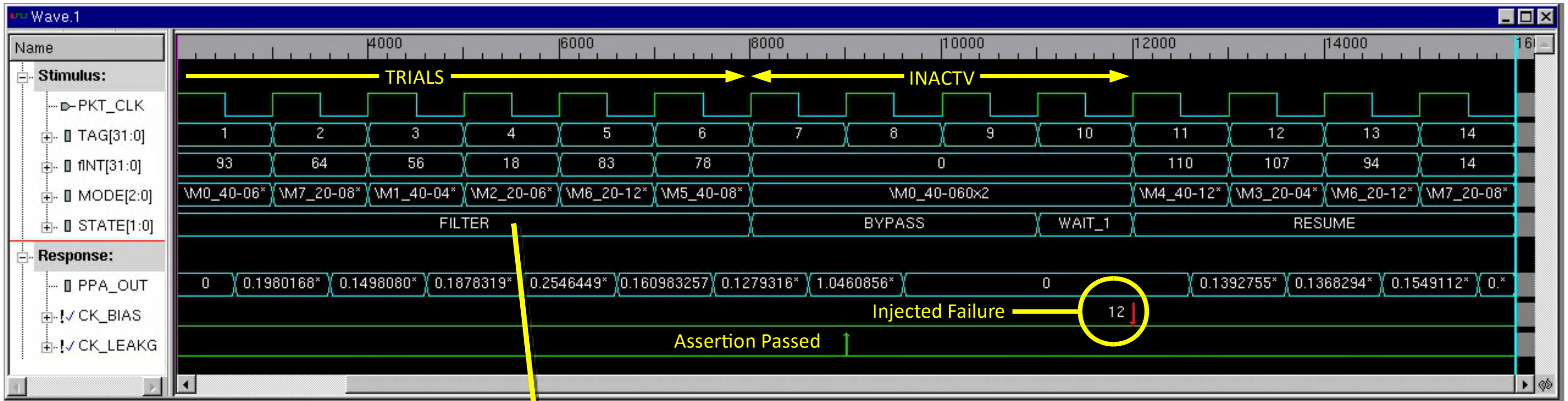

```
//Check power-down leakage current:
  property LEAKAGE_pro;
    @(posedge FREQ_IN.PKT_CLK)
    $rose(FREQ_IN.STATE == BYPASS) |->
      IDD < 10e-9; //Passing level.
  endproperty: LEAKAGE_pro
  .  .  .  .  .
  CK_LEAKG:
  assert property (LEAKAGE_pro)
    uvm_report_info("PROPS",
      «formatted string», UVM_HIGH
    );
  else //Warn at verbosity UVM_NONE:
    uvm_report_warning("PROPS", …);
```



Guideline:
Simplest approach: put assertion code in fixture, closest to I and V.

- Concurrent assert statement checks named property.
- Evaluates its condition at every edge of specified clock.
- Print pass/fail results using UVM methods, by severity.

Test-Suite Waveforms



+TRIALS=6
+INACTV=4

Optional State Aids Analysis

Guideline:
UVM factory substitution enables users to inject an error test instead of default—without altering or recompiling code.

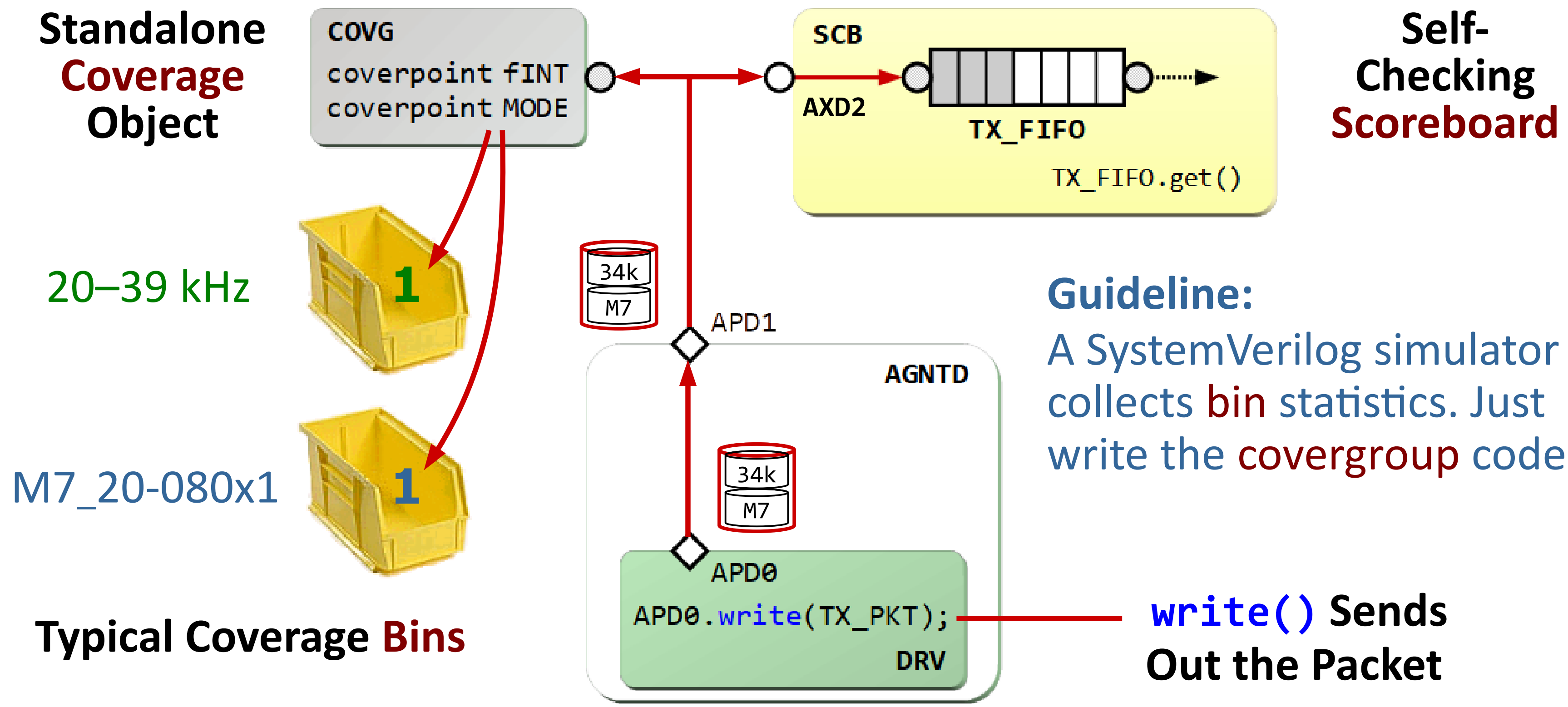
- Extended test suite signals displayed in DVE wave view.
- Property CK_LEAKG passed at end of first BYPASS cycle.
- Due to deliberately-injected error, CK_BIAS has failed.

§9. Functional Coverage

- The Coverage Object
- Basic Covergroup Code
- Extended Covergroup Code
- When is Data Sampled?
- Hitting 100% Coverage

Slide count: 5 of 57

The Coverage Object



- We drove random frequencies and modes into the DUT.
- Did our test suite cover all possible `fINT`, `MODE` values?
- Uncovered features could conceal lurking design bugs.

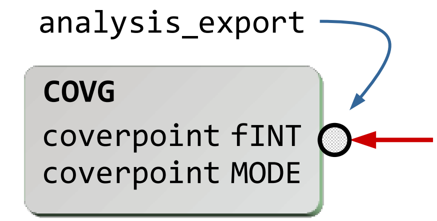
Basic Covergroup Code

Coverpoint Name in Database

```
class COVERAGE extends uvm_subscriber #(PACKET);
  .   .   .   .   .
  PACKET TX_PKT;
  covergroup CVG; //Cover applied frequency:
    FREQ_cvg: coverpoint TX_PKT.fINT
      {                //   kHz
        bins B10  = {[ 10: 19]};
        bins B20  = {[ 20: 39]};
        bins B40  = {[ 40: 59]};
        bins B60  = {[ 60: 79]};
        bins B80  = {[ 80: 99]};
        bins B100 = {[100:120]};
      }
  endgroup: CVG
  .   .   .   .   .
endclass: COVERAGE
```

Subdivide 10–120 kHz Range Into Six Bins

Standalone Class in Environment



uvm_subscriber Component

Bins for FREQ_CVG

NAME	COUNT	AT LEAST
B100	5	1
B80	3	1
B60	2	1
B40	4	1
B20	5	1
B10	1	1

TRIALS = 20

- One or more covergroups tells **VCS** to track coverage.
- Generates coverage database in directory `./simv.vdb`.

Extended Covergroup Code

```
covergroup CVG;
  FREQ_cvg: coverpoint TX_PKT.fINT
    {                //   kHz
      bins B10  = {[ 10: 19]};
      bins B20  = {[ 20: 39]};
       . . . .      . . . .
      bins B100 = {[100:120]};
    }
  //Cover applied mode values:

  //Cover combinations:  MODE x fINT

endgroup: CVG
```

Cartesian Cross-Product

Guideline:

Prefer enumerated control values. Binned automatically—based on cardinality of set.

MONITOR#(40,5)

VARIABLE	COVERED	%
FREQ_cvg	6	100.00
MODE_cvg	8	100.00

CROSS	COVERED	NOT	%
CROSS_cvg	31/48	17	64.58

TRIALS = 40

- A covergroup comprises one or more coverage points.
- Each coverpoint can refer to a variable or expression.
- Can also specify cross-coverage between cover points.

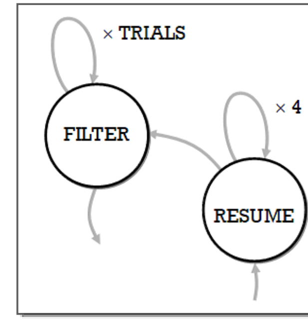
When is Data Sampled?


```
/* The driver's APD0.write(TX_PKT)
 * method broadcasts TX_PKT to all
 * subscribers. In turn, it will call
 * their uvm_subscriber write() method.
 */
virtual function void write(PACKET t);
  TX_PKT = t;   //Copy PKT pointer.
  if (TX_PKT.STATE == FILTER ||
      TX_PKT.STATE == RESUME
   )
    CVG.sample();
endfunction: write
```

COVERAGE Class

Sample the Data at Broadcast

No Sampling In Inactive States

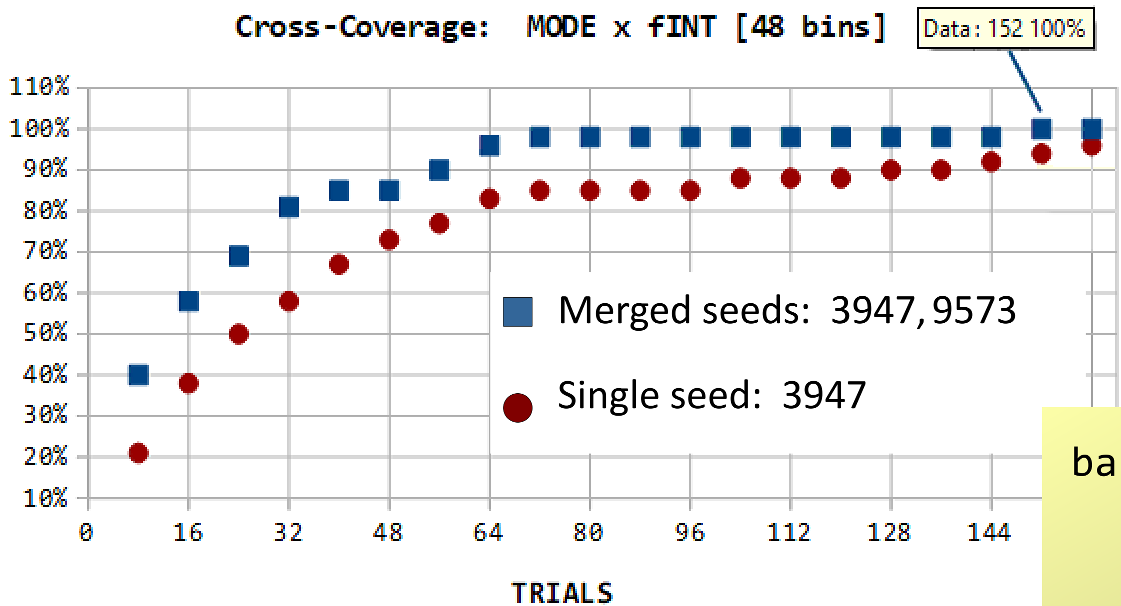


Guideline:
Avoid altering name *t* of argument in virtual method of base class.

Cummings & Chambers,
UVM Analysis Port Functionality, Austin SNUG-2018, §2. Observer Pattern, p. 14

- Can specify clocking event at which coverage sampled.
- Or procedurally trigger sampling: call `CVG.sample()`.
- Must invoke it only during the normal filtering states.

Hitting 100% Coverage



Uncovered bins

MODE_cvg	FREQ_cvg	COUNT
[auto_M1_40-040x2]	[B40]	0
[auto_M2_20-060x2]	[B100]	0

TRIALS = 64

```
bash> urg -dir simv1.vdb  \
          -dir simv2.vdb  \
          -dbname merge.vdb
```

VCS Benchmark Runs

VCS Unified Report Generator Script

- Covering eight modes is easy. But what about MODE × f?
- Probability of hitting any one of 8x6=48 bins is just 2%.
- Reduce TRIALS by merging runs for two random seeds.

§10. Wrap-Up

- UVM Takeaway Points
- XMODEL Takeaway Points
- For More Information

UVM Takeaway Points

1. All UVM components/objects derive from base classes.
2. Communicate by sending packets over TLM pathways:

```
AGNTD.APD1.connect(SCB.AXD2);
```

APD1 → AXD2

3. Interface buses connect the component hierachy with the DUT via uvm_config_db database settings:

{Key_VMIF: MIF}

```
uvm_config_db #(VIF_t)::get(
  this, "", "Key_VMIF", VMIF);
```

4. Topmost module calls run_test() at time 0 to run suite:

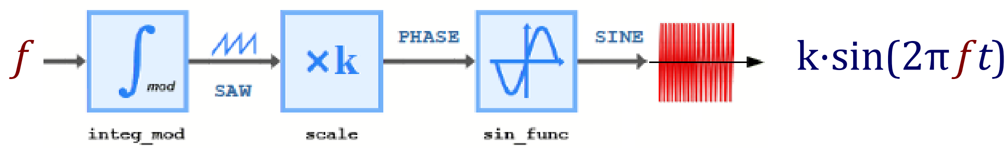
```
initial run_test("TEST_SUITE");
```

TEST_SUITE

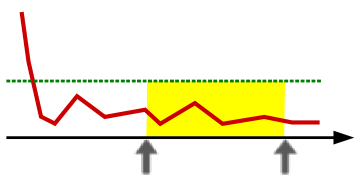
5. UVM automates the successive phases in a simulation.

XMODEL Takeaway Points

6. XMODEL: plug-in extension to SystemVerilog simulators.
7. Rich library of analog and digital primitive elements:



8. Analog assertions check features even in power-down:



```
CK_LEAKG: assert property (LEAKAGE_pro)
    uvm_report_info(«pass action»);
```

9. Seamlessly compatible with full SystemVerilog syntax:

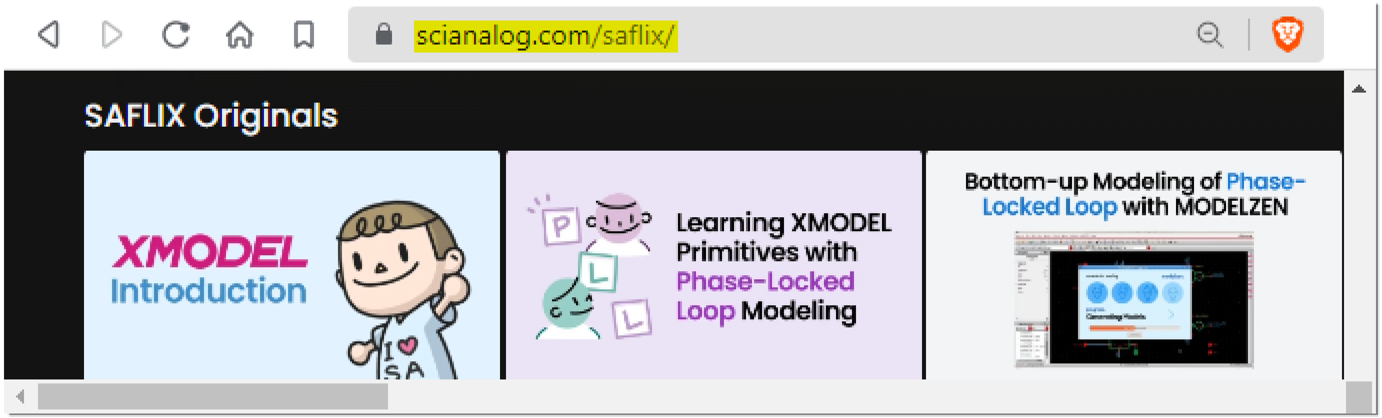
```
FREQ_cvg: coverpoint TX_PKT.fINT
  { bins B10 = {[ 10: 19]}; ... }
```

VARIABLE	COVERED	%
FREQ_cvg	6	100.00
MODE_cvg	8	100.00

10. SPICE-like accuracy at nearly logic-simulation speeds.

For More Information

For technical videos, tutorials, and application demos, visit: `scianalog.com`



To download a PDF copy of the webinar, and the UVM testbench code:
www.scianalog.com/webinars/w20220621

For more coding details on XMODEL primitives, scoreboard methods, assertions:
C. Dančak, “SystemVerilog OOP Testbench, Parts {1,2}:
www.researchgate.net/publication/
346061868_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_1
350412143_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_2