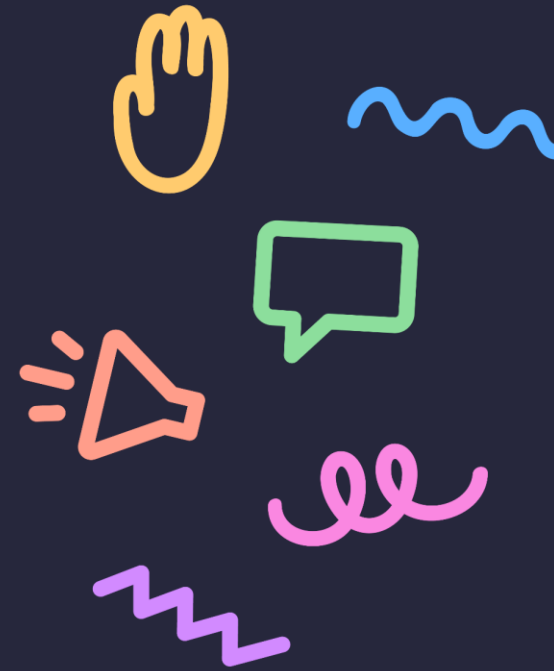


X Webinar Series on **ANALOG VERIFICATION INSIGHTS**



**Mixed-Signal Verification of Analog IP
using SystemVerilog:
An Object-Oriented Approach**



Charles Dančák University of California, San Diego | UCSD • Extension

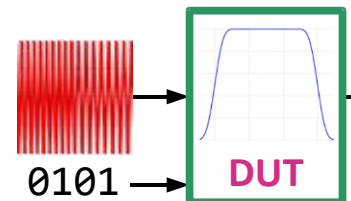
scientific
analog

Mixed-Signal Verification of **Analog** IP using SystemVerilog: An Object-Oriented Approach

Scientific Analog, Inc.

June 2021

```
//Random frequency:  
rand int fINT; //kHz.  
//Random mode bits:  
randc MODE_t MODE;
```



Contents

1. Testbench for Analog
2. Fixture Subcircuits
3. Sequence Component
4. Driver and Monitor
5. Scoreboard Component
6. Bypass/Power-Down
7. Analog Assertions
8. Functional Coverage
9. Environment Module
10. OOP/XMODEL Guidelines

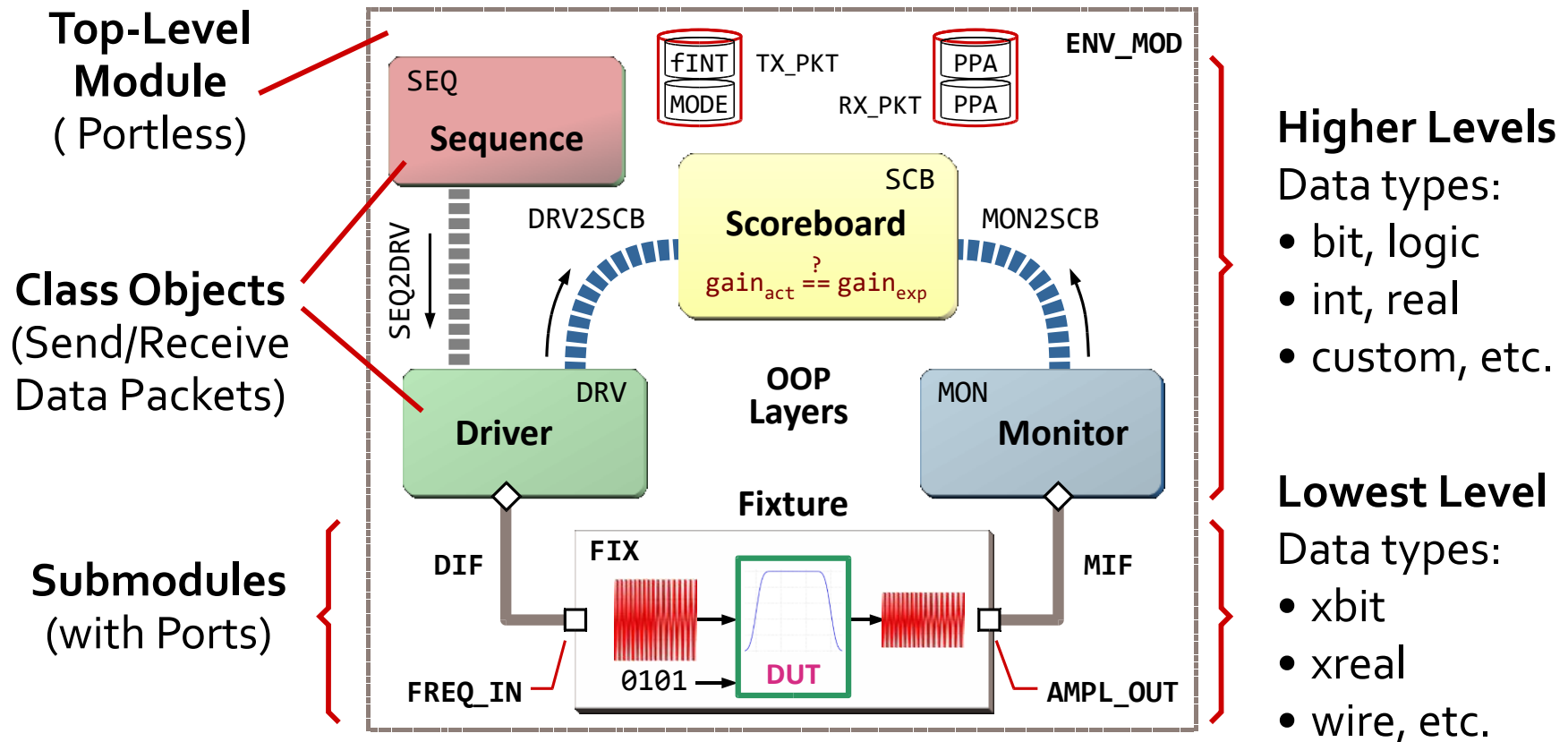
§1: Testbench for Analog

- OOP Testbench Organization
- A Mixed-Signal Filter DUT
- Bandpass Filter Architecture
- Analog Modes in SystemVerilog
- Filter Verification Plan
- A Packet per Transaction
- Interface Bus Description
- FAQs to be Answered

Note:

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

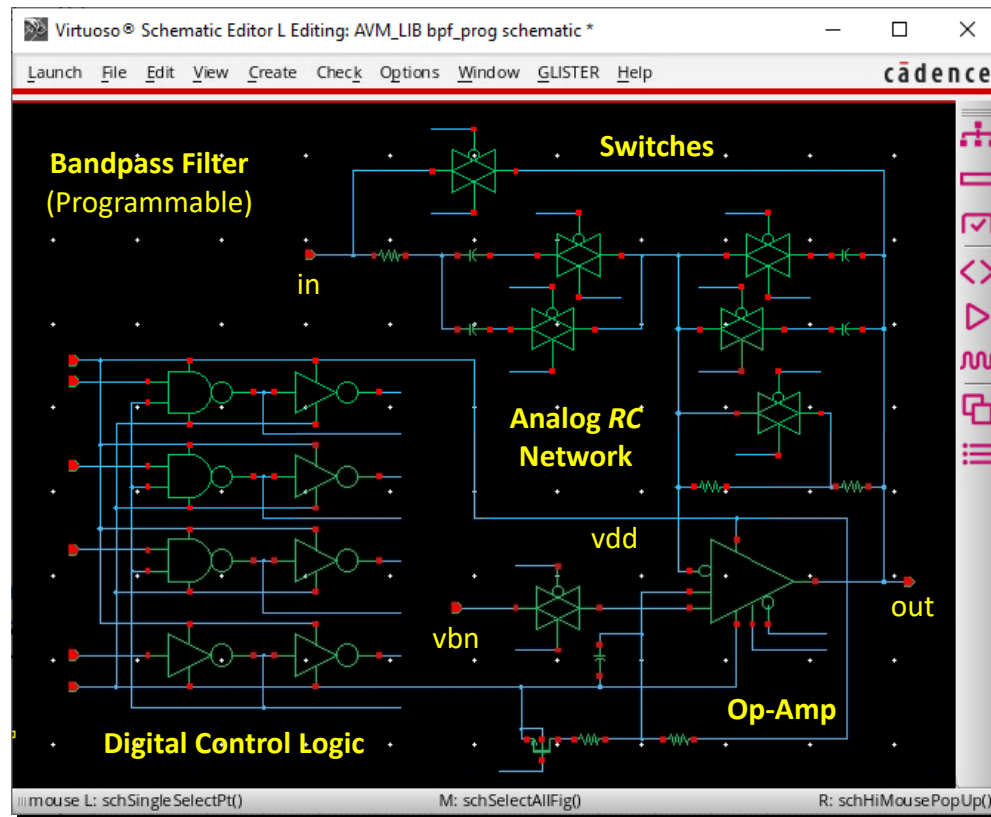
OOP Testbench Organization



- High-level components in OOP testbench are **objects**.
- **Packets** contain transaction data (stimulus, response).

A Mixed-Signal Filter DUT

**Programmable
Automotive
Bandpass Filter**
(10–120 kHz)



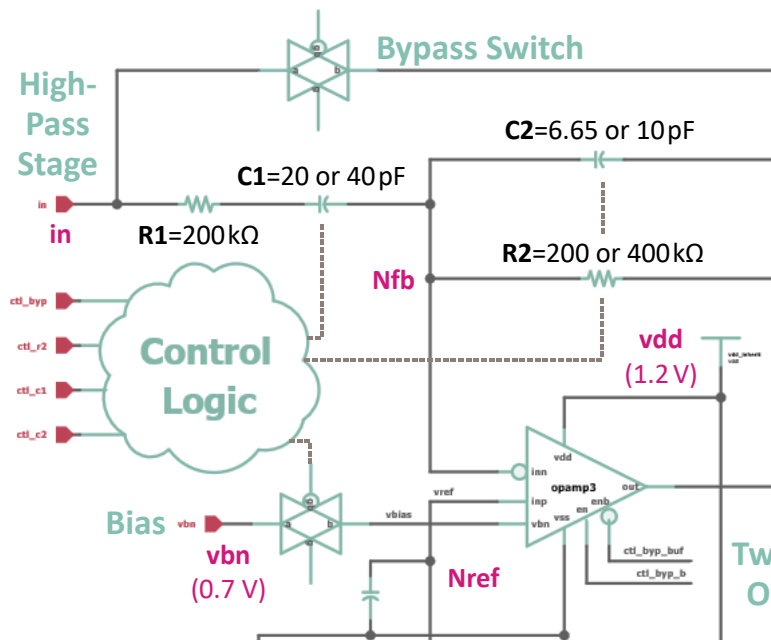
**GLISTER
Toolbar**
(Requires
XMODEL
Plug-In)

- Active *RC* bandpass filter, with eight passband modes.
- Designed in **Virtuoso**, with transistors from 45-nm PDK.

Bandpass Filter Architecture

Digital
Control
Word

ctl_byp
ctl_r2
ctl_c1
ctl_c2



Analog
RC Network

Bypass/Power-Down:

- Input feeds output.
- Op-amp tristated.
- Supply is cut off.

Dashed Lines
Indicate
Digitally-Controlled
Switch Circuitry

- Four-bit control input **selects** 1 of 8 modes—or bypass.
- These bits switch CMOS **transmission gates** on and off.
- Puts ***R***, ***C*** elements in series or shunt, to vary **passband**.

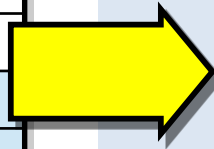
Filter Verification Plan

Applied Stimulus	Quantity	Random?	Monitored Response
Input Frequency	$f=10\text{--}120\text{ kHz}$	uniform	Measure: $\text{gain}_{\text{act}} == \text{gain}_{\text{exp}}$
Passband Mode	\M0..\M7	cyclic	
Frequency \times Mode	Cross coverage	yes	100% of $6 \times 8 = 48$ bins
Bypass/Pwr-Down	Supply Idd	no	Assert: Idd $\leq 5\text{ nA}$
Power-On Reset	Bias Vbn	no	Assert: Vbn = $700 \pm 50\text{ mV}$

- Each transaction applies **randomized inputs** to filter.
- Both a random **frequency** and a cyclic-random **mode**.
- Filter's **gain** is then checked versus a reference model.
- Testbench checks supplemented by **analog** assertions.

Analog Modes in SystemVerilog

Digital Control Word: {ctl_byp, ctl_r2, ctl_c1, ctl_c2}				
Binary Value	Enumeration Literal	f_{LO}	f_{HI}	g
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	—	—	—



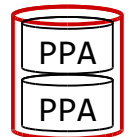
```
//Enumerated filter modes:
typedef
enum bit [2:0] {
  //Filtering modes:
  \M0_40-060x2 = 3'b000,
  \M1_40-040x2 = 3'b001,
  . . . . .
  \M6_20-120x1 = 3'b110,
  \M7_20-080x1 = 3'b111
} MODE_t;
```

Escaped Name

- An **enum** type captures these **modes**—with **encodings**.
- Custom **typedef** is packaged; imported where needed.
- High-level OOP code can now refer to a mode **by name**.

A Packet per Transaction

Packet Field	Data Type	Used In	Range, Units
TAG	int	TX, RX	≥ 1
fINT	int	TX, RX	10–120 kHz
fREAL	real	TX	
MODE	MODE_t	TX, RX	\M0..\M7
PPA_IN	real	RX	≥ 0.00 Volts
PPA_OUT	real	RX	



- **Tags** ensure that we compare n^{th} TX with n^{th} RX packet.
- **Driver** applies the packet's fields to input pins of DUT.
- **Monitor** samples DUT output pins; reassembles packet.


The Packet Object (1/2)

```

class PACKET;
    int TAG = 0; //Packet ID.
    //Random passband mode (M0..M7):
    rand MODE_t MODE;
    //Random frequency (10--120 kHz):
    rand int fINT;    //kHz.
    real    fREAL;    // Hz.
    //Constrain fINT to a range:
    constraint fRANGE_con {
        fINT inside { [10:120] };
    }
    . . . .
endclass: PACKET

```

TX_PKT
(From DRV to SCB)



Random Stimulus → (points to `rand MODE_t MODE;` and `rand int fINT;`)

Constraining the Random Frequencies → (points to `constraint fRANGE_con { fINT inside { [10:120] }; }`)

Reference:
Cummings, *UVM Transactions* (SNUG-14) §3.2

- A packet is **one transaction**, with stimulus and response.
- **Same** packet class **reused** for both **TX_PKT** and **RX_PKT**.

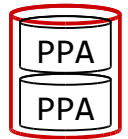
The Packet Object (2/2)

Peak-Peak
Input and
Output

```
class PACKET;
    • • • •
    //Measured peak-peak amplitudes:
    real PPA_IN, PPA_OUT; //Volts.
```

Random int
Cast to real

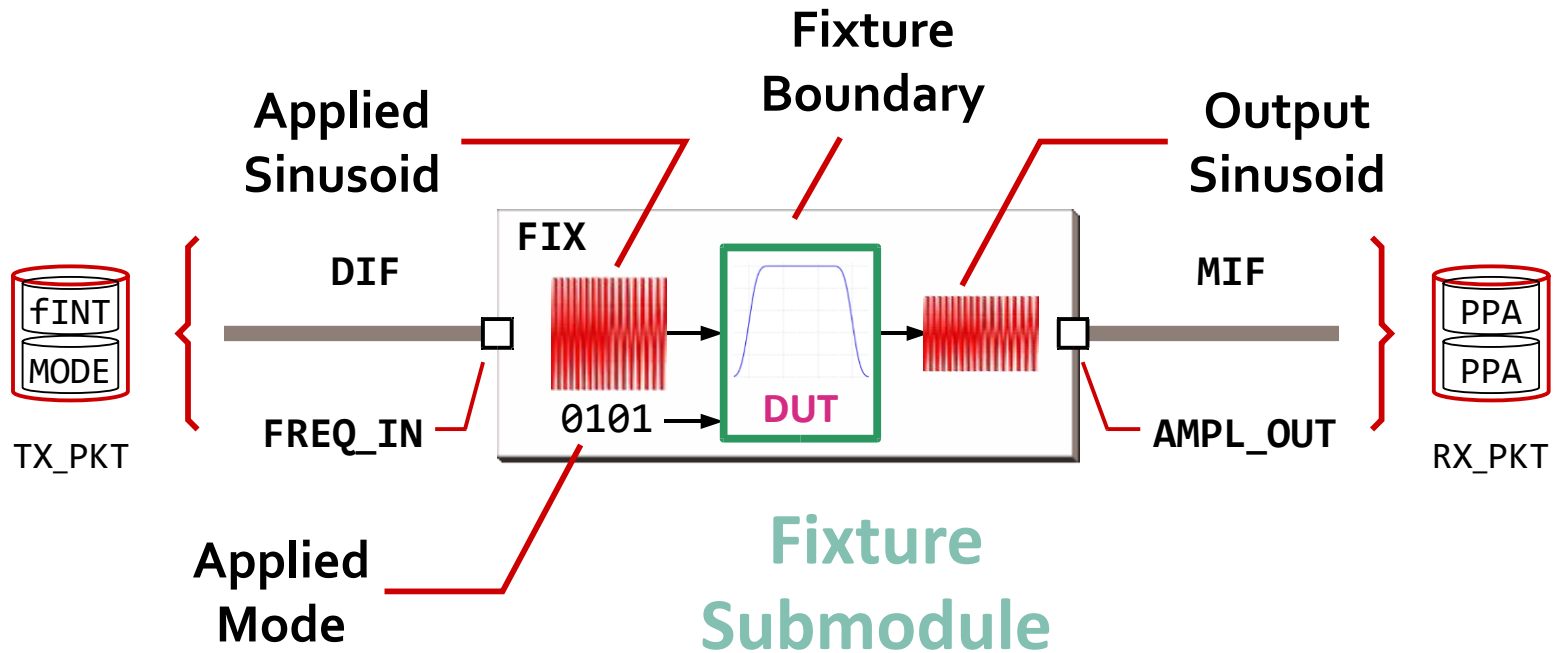
```
    //Called after .randomize():
    function void post_randomize();
        • • • •
        //Cast random fINT to a real:
        fREAL = real'(fINT * 1e3);
    endfunction: post_randomize
endclass: PACKET
```



RX_PKT
(From MON
to SCB)

- Constraints on frequency, mode **shape** random stimuli.
- Can only randomize an **integer-valued** frequency f_{INT} .
- Post-randomize conversion: e.g. **20 kHz** to 20,000.0 Hz.

Interfacing with the DUT

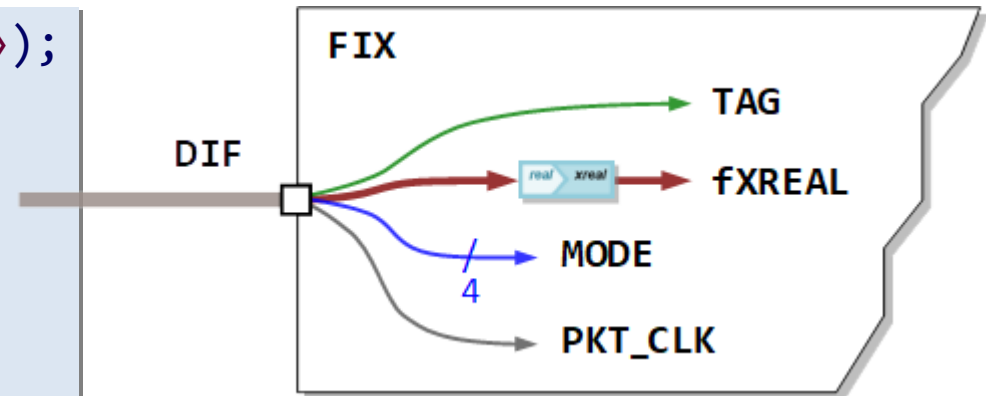


- **Sequence** object sends a transaction packet to driver.
- Driver sends fields of TX_PKT over interface bus, **DIF**.
- **DIF** is a **virtual** interface, able to connect to an object.
- Monitor gets fields over **MIF**; reassembles into RX_PKT.

Interface Bus Description

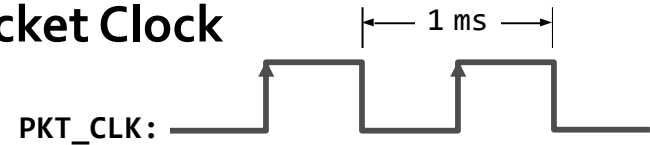
```
interface BAND_IF(«clock...»);
  import GOLD_PKG::*;

  //Packet tag:
  int TAG;
  //Enumerated mode:
  MODE_t  MODE;
  //Random frequency:
  int fINT; real fREAL;
  . . . .
  //Peak-peak amplitudes:
  real PPA_IN, PPA_OUT;
endinterface: BAND_IF
```



```
//Instantiated in ENV_MOD:
BAND_IF DIF(PKT_CLK, RST);
```

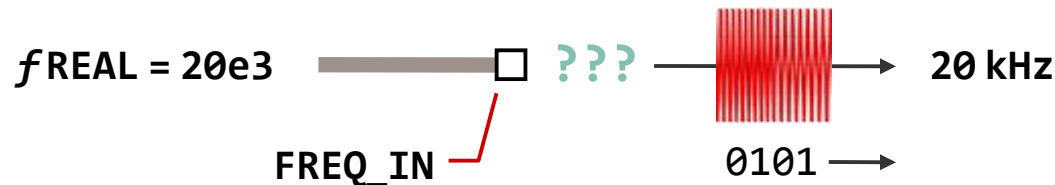
Packet Clock



- An **interface**: a **named bundle** of signals of any type.
- Includes 1-ms **PKT_CLK** from the environment module.
- Bus instance DIF carries packets **into** fixture, and DUT.

FAQs to be Answered

- A. How is the SystemVerilog real frequency f^{REAL} converted to a **sinusoidal** input to an analog filter?

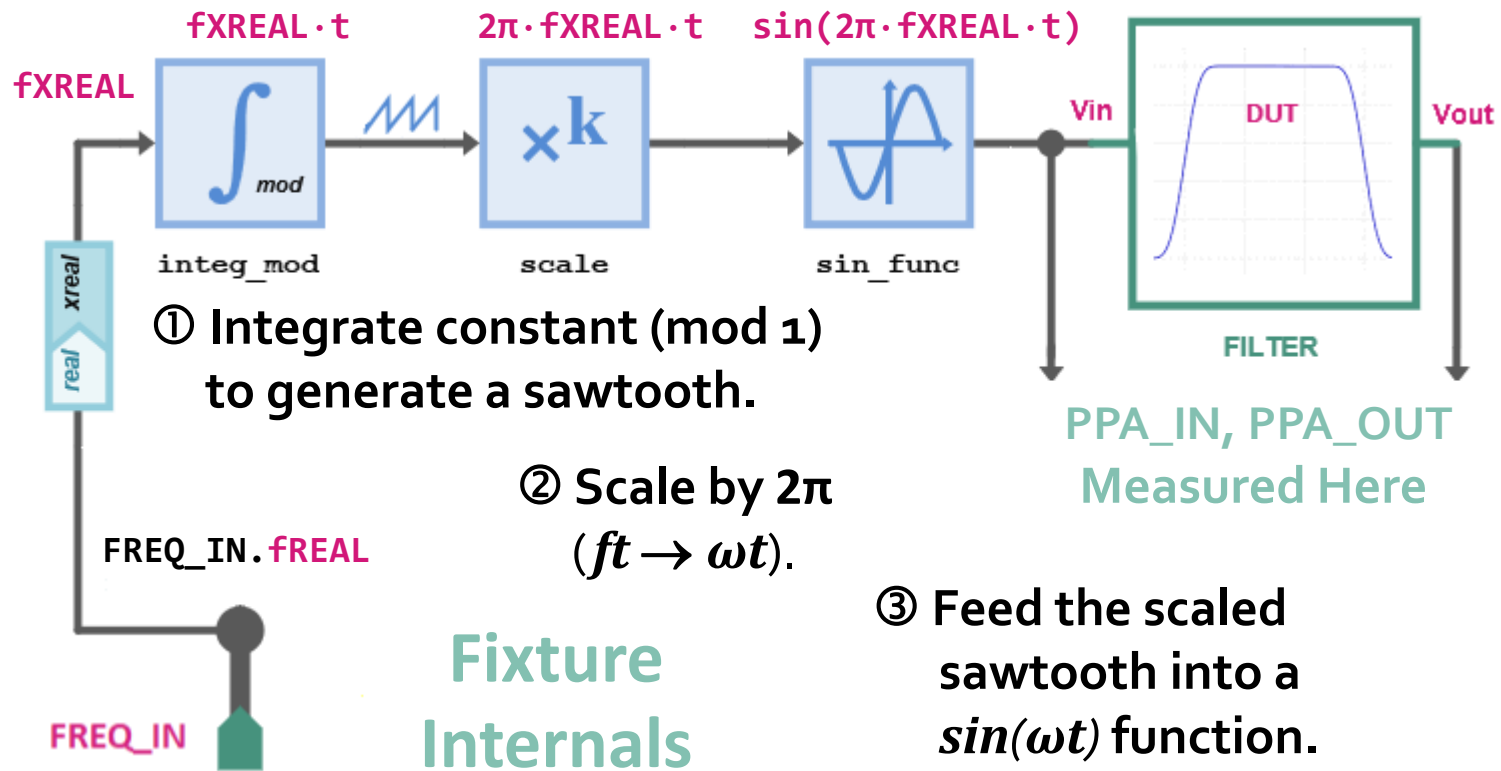


- B. How can we measure the **peak-to-peak amplitude** of the filter's sinusoidal input and output—especially over a time span after transients have died out?
- C. What enables a **logic simulator** like **VCS** or **Xcelium** to accurately simulate the filter's **analog** behavior?

§2: Fixture Subcircuits

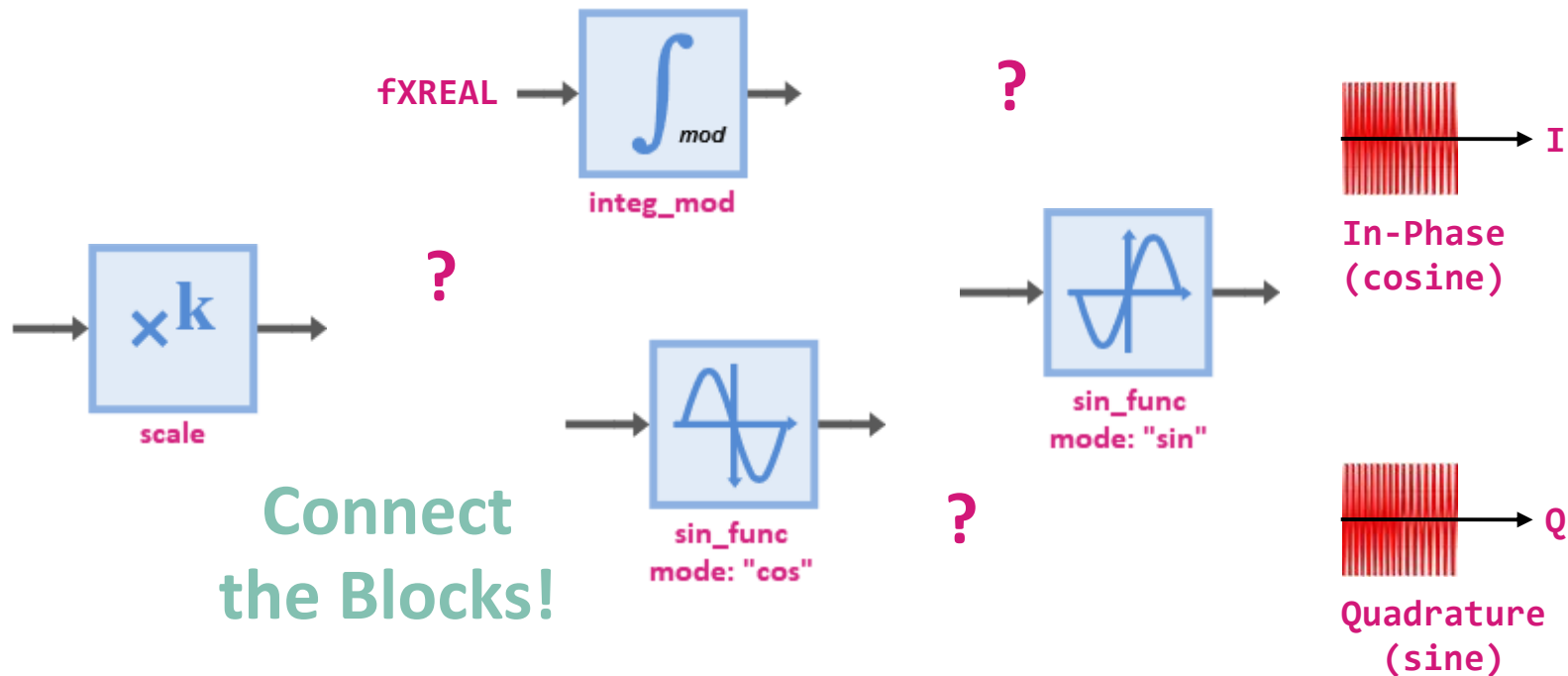
- A Frequency Subcircuit
- Exercise #1
- A Mode Subcircuit
- Cyclic Random Modes
- Measure Time-Varying PPA
- A PPA Subcircuit
- Preview: Actual Gain
- Fixture Summary

A Frequency Subcircuit



- Real input frequency value is converted to **xreal** type.
- Chain of several primitives generates a **sine** stimulus.
- Type **xreal** enables event-driven simulation of **analog**.

Exercise #1: Quadrature Signals

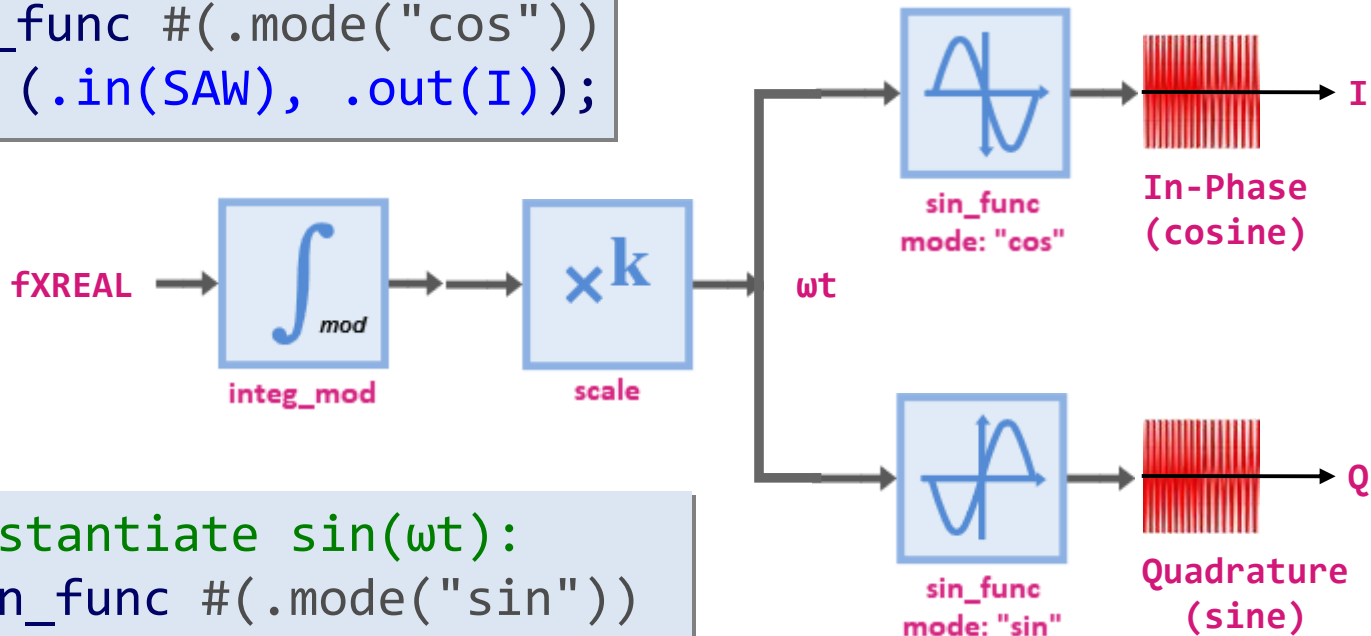


Connect
the Blocks!

- Connect the primitives to generate **quadrature** signals.
- Outputs are a **sine** and **cosine**, of the same frequency.
- Hint: **sin_func** has a **mode setting** of "sin" or "cos".

Answer #1: Quadrature Signals

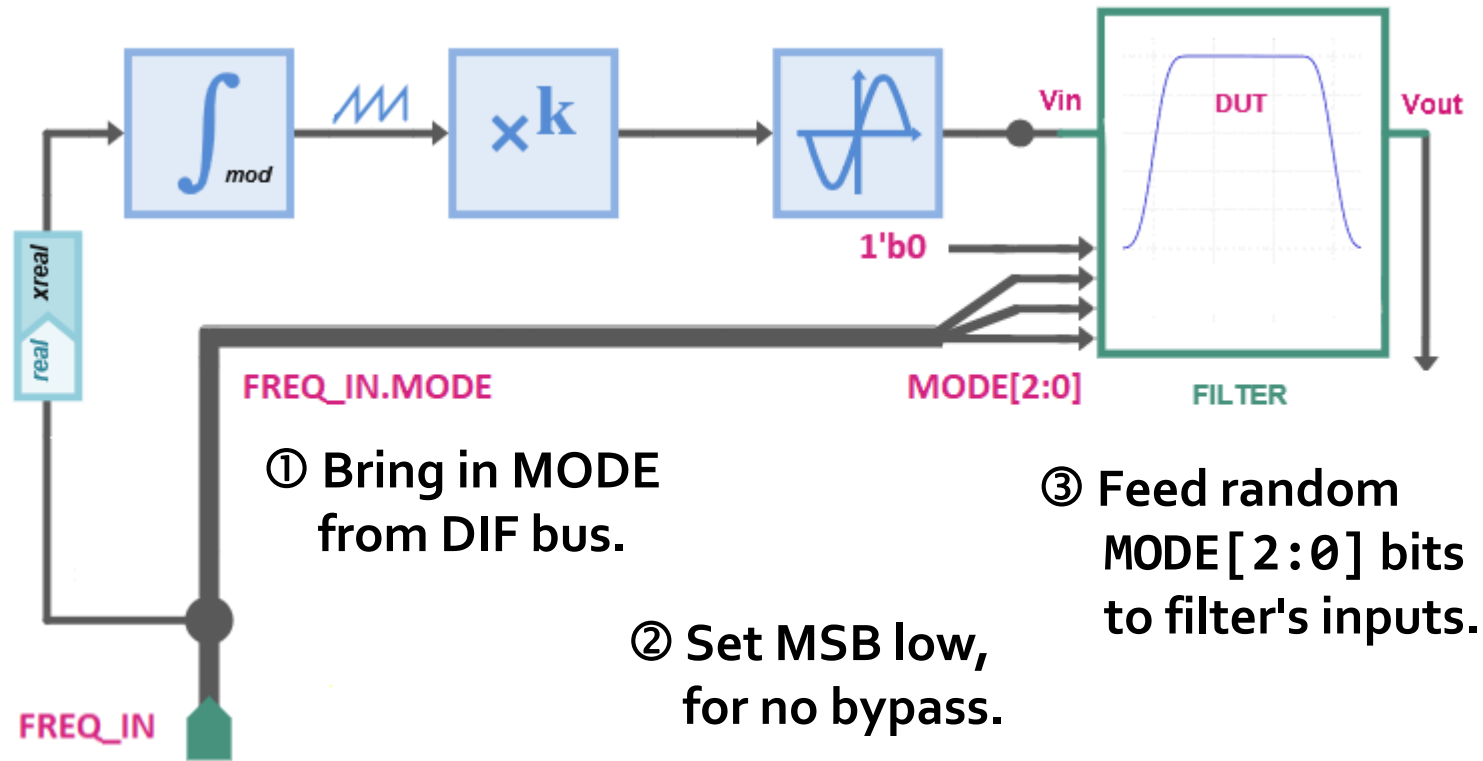
```
//Instantiate cos( $\omega t$ ):
sin_func #(.mode("cos"))
F0 (.in(SAW), .out(I));
```



```
//Instantiate sin( $\omega t$ ):
sin_func #(.mode("sin"))
F1 (.in(SAW), .out(Q));
```

- Can place elements from XMODEL library in **Virtuoso**.
- Or **instantiate** the elements into SystemVerilog code.

A Mode Subcircuit



- Let's defer any bypass/power-down testing until later.
- Set MODE[3] bit low for now, to disable bypass mode.
- Apply the randomized MODE[2:0] to the filter's inputs.

Cyclic Random Modes (1/4)

No Modes
Repeated
in a Cycle

New
Cycle
Starts

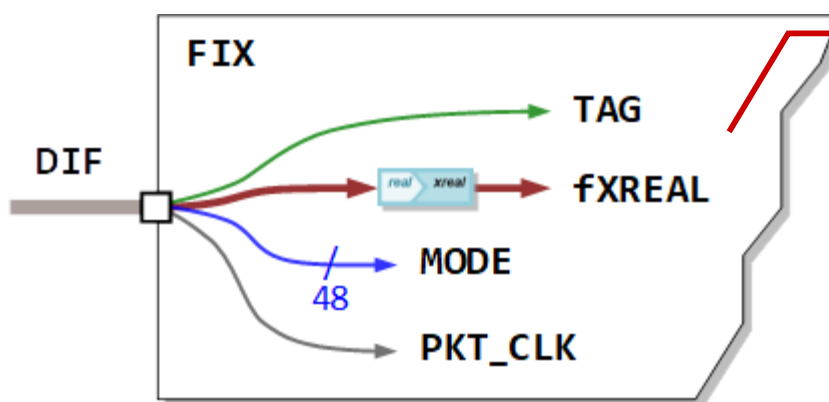
```
Launch SEQ.BODY...
#1. M4_40-120x1 at 21 kHz
#2. M0_40-060x2 at 120 kHz
#3. M7_20-080x1 at 19 kHz
#4. M6_20-120x1 at 117 kHz
#5. M3_20-040x2 at 103 kHz
#6. M1_40-040x2 at 91 kHz
#7. M2_20-060x2 at 32 kHz
#8. M5_40-080x1 at 21 kHz
#9. M3_20-040x2 at 60 kHz
. . . . .
```

```
class PACKET;
    int TAG = 0;
    //Random cyclic:
    randc MODE_t MODE;
    . . . . .
endclass: PACKET
```

Declare a **Cyclic**
Random Variable

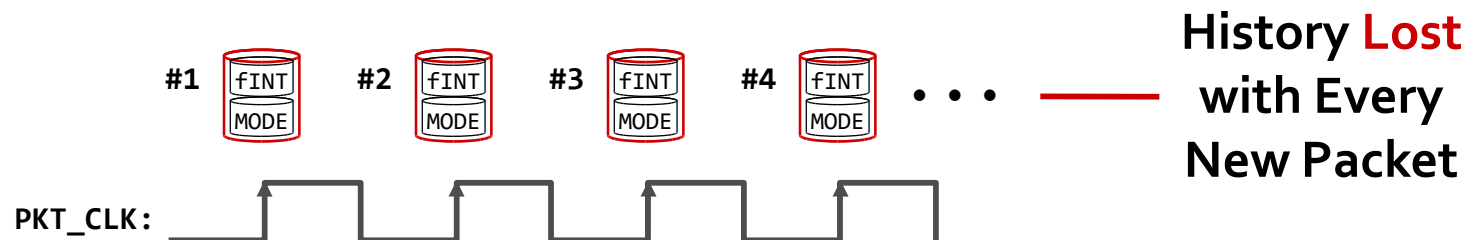
- Ideally, we'd like randomized **MODE** values to be **cyclic**.
- In a cycle of eight values, **no repetitions** would occur.
- A **randc** qualifier automatically yields cyclic behavior.

Cyclic Random Modes (2/4)



Scaling MODE to **48** Bits

ERROR [RCNAOLV]:
Random Cyclic Not Allowed on Long Vectors
 Variable **MODE** is 48 bits wide, and can't be declared as **randc**. Maximum supported bit width in this tool is: 32.



- But **randc** is limited: lacks **scalability** and **persistence**.
- Simulators may restrict **bit width** of a **randc** variable.
- Cyclic behavior won't persist from **one** packet to **next**.

Cyclic Random Modes (3/4)

```
class PACKET;
  • • • •
  //Random mode (M0..M7):
  rand MODE_t MODE;

  //Queue, storing used values:
  static MODE_t USED[$] = '{}';

  //Constrain MODE to unused:
  constraint XCLUDE_con {
    !(MODE inside {USED});
  }
  • • • •
```

Empty Initial Queue

```
//Post-process the queue:
function...post_randomize();
  • • • •
  //Append till almost full:
  if (USED.size < 7)
    USED.push_back(MODE);
  else
    //Last mode not pushed:
    USED.delete(); //Empty.
  • • • •
endfunction: post_randomize
endclass: PACKET
```

- Workaround: save the history with a **queue** of MODE_t.
- **Static** queue keeps track of MODE values used thus far.
- Next mode chosen is **constrained** to values yet unused.

Cyclic Random Modes (4/4)

```

$time  MODE  .size  Queue USED[$]
-----
1 ms:   2      1    {'h2}
2 ms:   3      2    {'h2, 'h3}
3 ms:   6      3    {'h2, 'h3, 'h6}
4 ms:   4      4    {'h2, 'h3, 'h6, 'h4}
5 ms:   7      5    {'h2, 'h3, 'h6, 'h4, 'h7}
6 ms:   5      6    {'h2, 'h3, 'h6, 'h4, 'h7, 'h5}
7 ms:   1      7    {'h2, 'h3, 'h6, 'h4, 'h7, 'h5, 'h1}
8 ms:   0      0    {}
9 ms:   5      1    {'h5}
10 ms:  3      2    {'h5, 'h3}
. . . . .

```

Dave Rich (Siemens/Mentor), post 5015,
Solution: randc Restrictions, 9 Jan 2013,
[verificationacademy.com/forums/
 systemverilog/randc-problem](http://verificationacademy.com/forums/systemverilog/randc-problem)



- Simulation transcript shows queue **grow** and **shrink**.
- At depth 7, only mode left is 0. **Empty** queue, restart

Measure Time-Varying PPA

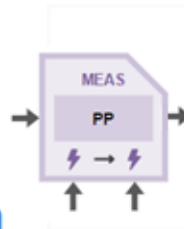
PRIMITIVES / MEASUREMENT PROBES / MEAS_PP

meas_pp :

A primitive for measuring the peak-to-peak value of an xreal-typed signal within a time interval.

Input/Output Terminals

Name	I/O	Type	
out	output	real	
in	input	xreal	
from	input	xbit	
to	input	xbit	to trigger



On-Line Data Sheet
(At [Support Website](#))

```
//Measure peak-to-peak sine input:
meas_pp M_PRE(
    .in(SINE),.out(AMPL_OUT.PPA_IN),
    .from(TICK_x), .to(TOCK_x)
);
```

Real-Valued Output

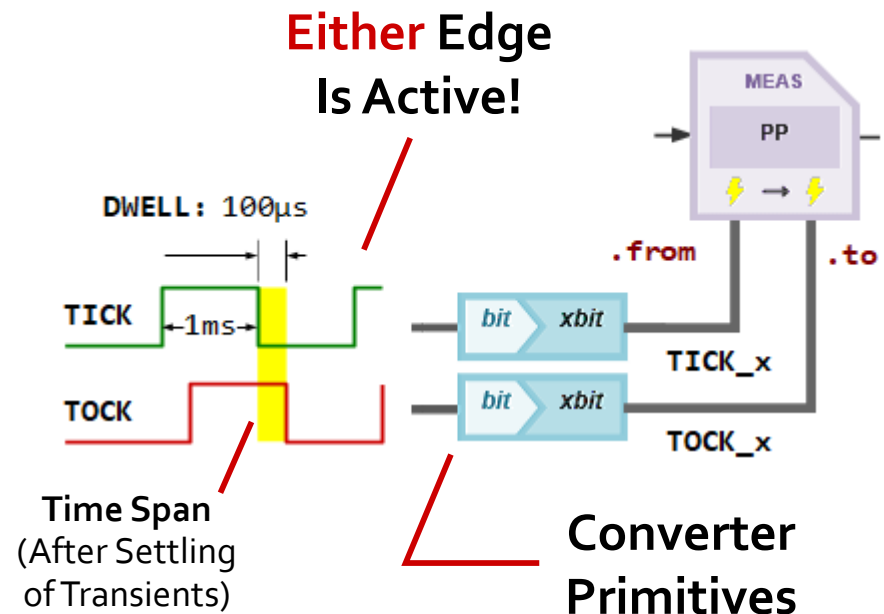
- Place in **Virtuoso**, or instantiate in SystemVerilog code.
- Data sheet describes pin names, with their **data types**.
- Measurement primitives have output types like **real**.

Delimiting a Time Span

```

initial
begin:MEASURE
  «wait till after reset»
  forever
    begin:TICKING
      @(negedge «PKT_CLK»);
      TICK = ~TICK;
      //Stop meas_pp after DWELL:
      #(DWELL) TOCK = ~TOCK;
    end: TICKING
  end: MEASURE

```



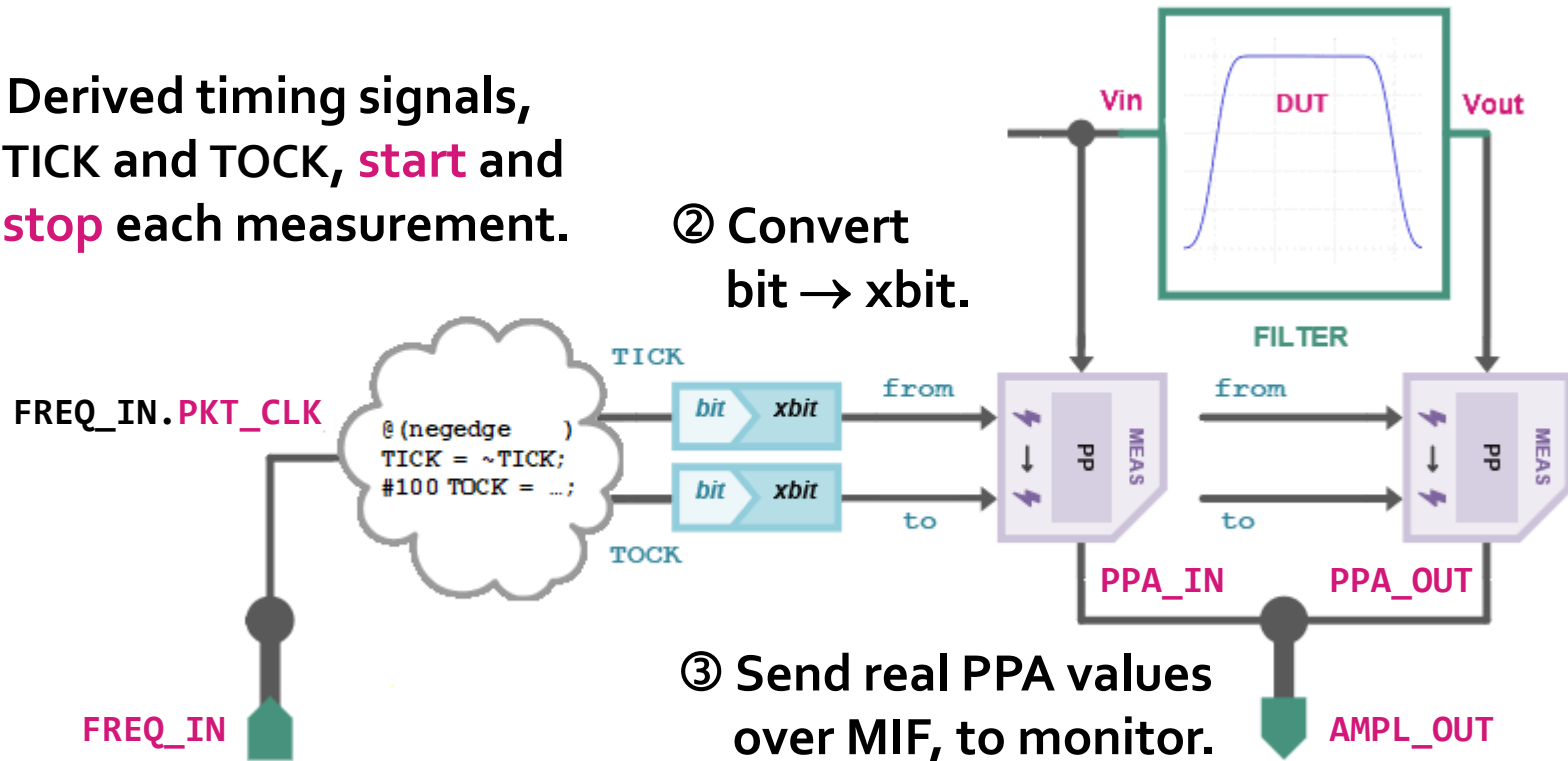
- Analog measurements often taken over **time intervals**.
- Interval delimited here by **trigger signals** of type xbit.
- Derived from PKT_CLK by a typical verification **code block**.
- Other **primitives** directly offer suitable xbit trigger output.

A PPA Subcircuit

① Derived timing signals, TICK and TOCK, **start** and **stop** each measurement.

② Convert bit → xbit.

③ Send real PPA values over MIF, to monitor.

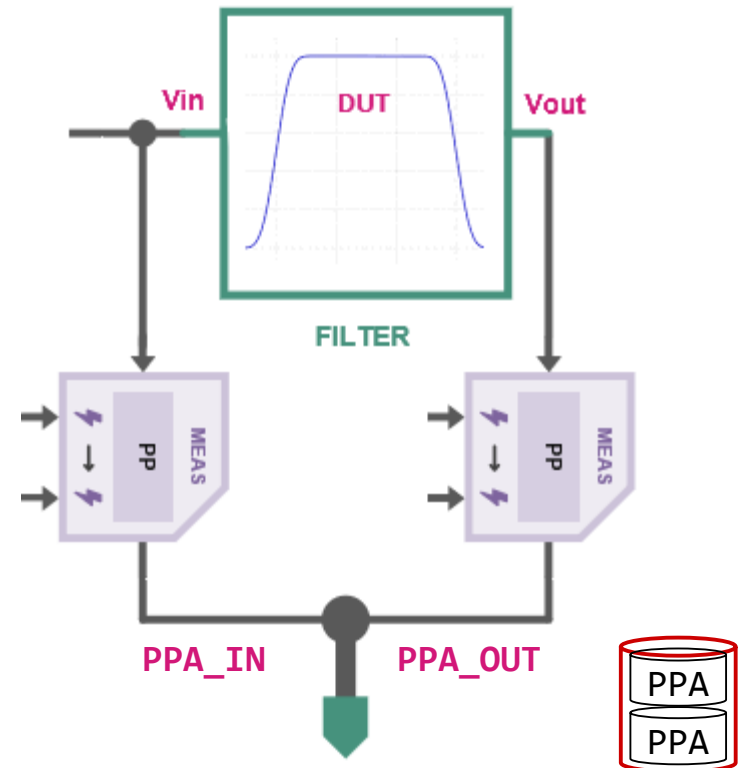


- Use **meas_pp** to measure amplitude over a time span.
- Triggering inputs **.from**, **.to** driven by **TICK_x**, **TOCK_x**.
- **Time span** between their triggering edges is 100 μ s.

Preview: Actual Gain

```
task SCORE_PACKET();
begin:SCORE_LOOP
    real gACTUAL, PPA_IN, PPA_OUT;
    /* Ratio of peak-peak values
     * [varies from ~0.3 to 1.5]:
     */
    PPA_OUT = RX_PKT.PPA_OUT;
    PPA_IN  = RX_PKT.PPA_IN;
    gACTUAL = PPA_OUT/PPA_IN;
    . . . .
```

Scoreboard Code



- Monitor packetizes this data; sends on to scoreboard.
- Scoreboard's task is to **compare** actual, expected gain.
- **Actual** gain is ratio of measured output/input amplitudes.

Summary: Fixture Module

- A. Submodule FIXTURE instantiates the **analog DUT**, with various XMODEL instrumentation subcircuits.
- B. Encapsulate all signals of type **xreal** or **xbit** inside the fixture. Use primitives such as **xreal_to_real** to convert them to and from types **real** and **bit**.
- C. Fixture can include any **SystemVerilog code** (e.g., `initial` block) to control timing, or process data.
- D. Structure types **xreal**, **xbit** enable analog and digital simulation on event-driven logic simulator—once an XMODEL plug-in has been installed.

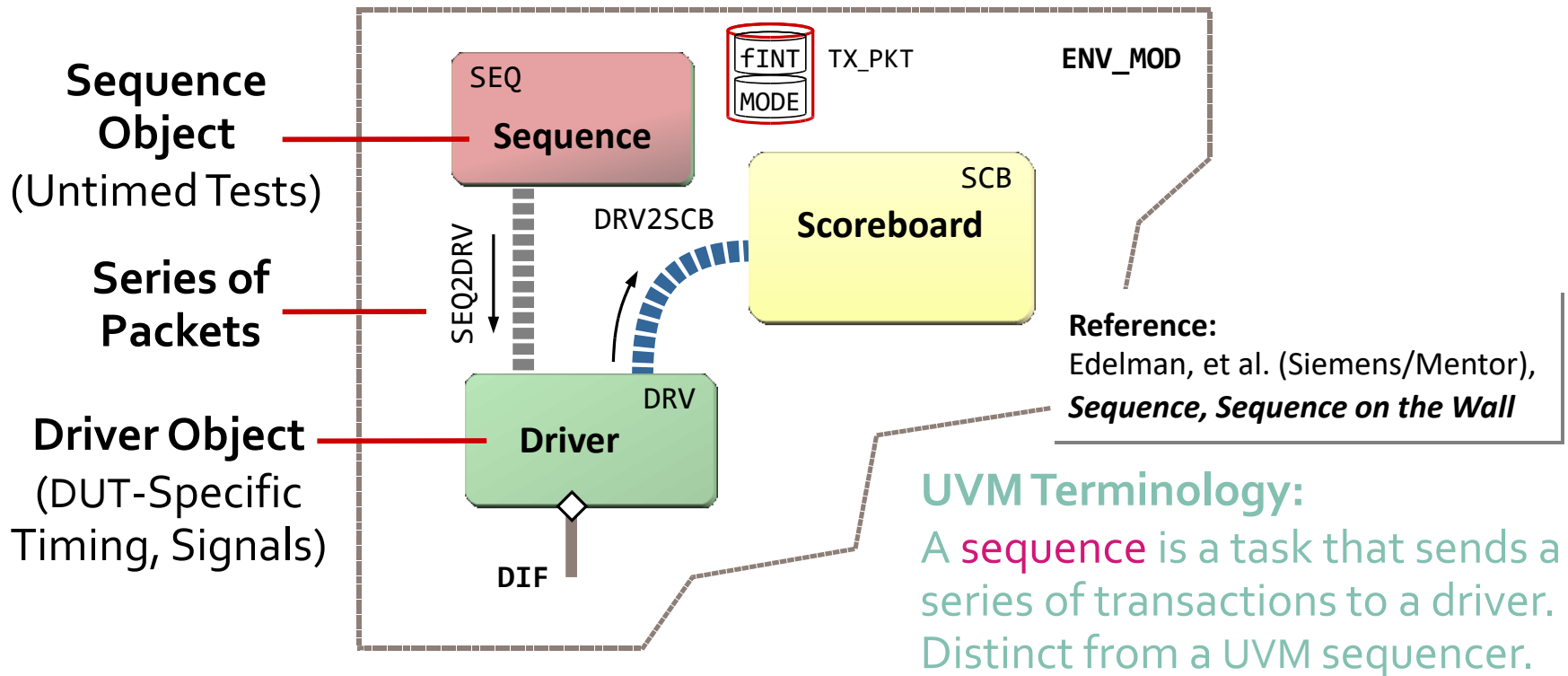
scientific
analog

29

§3: SEQ Component

- Sequence Architecture
- Basic Test Sequence
- Sequence BODY() Task
- A Simulated Sequence
- Exercise #2
- Sequence Summary

Sequence Architecture



- In Part 1, random stimuli came from the **driver** object.
- We now move the untimed stimuli to **sequence** object.
- Enhances **reuse**, and conforms more to UVM standard.

Basic Test Sequence

Applied Stimulus	Quantity	Random?	Monitored Response
Input Frequency	$f=10\text{--}120\text{ kHz}$	uniform	Measure: $\text{gain}_{\text{act}} == \text{gain}_{\text{exp}}$
Passband Mode	\M0..\M7	cyclic	
Frequency \times Mode	Cross coverage	yes	100% of $6 \times 8 = 48$ bins
Bypass/Pwr-Down	Supply Idd	no	Assert: $\text{Idd} \leq 5\text{ nA}$
Power-On Reset	Bias Vbn	no	Assert: $\text{Vbn} = 700 \pm 50\text{ mV}$

- A baseline plan: apply stimuli to the **powered-up** DUT.
- Later we extend the sequence to test **bypass/recovery**.
- **Sequence** object will **randomize** frequency and mode.
- **No timing** details, just a series of random transactions.

Sequence BODY() Task

```
class SEQUENCE
  #(int TRIALS = 16);
  //Stimulus to DRV:
  PACKET TX_PKT;
  //Queue to DRV:
  mailbox SEQ2DRV;
  . . .
  task BODY(); //UVM-like.
    for(«Loop for TRIALS»)
      begin:SEQ_LOOP
```

Class-Wide Parameter

New
TX_PKT



Debug
Aid

```
TX_PKT = new();
TX_PKT.TAG = I;
TX_PKT.randomize();
SEQ2DRV.put(TX_PKT);
$write(
  "%t ", $realtime,
  «key packet data»,
  "\n");
end: SEQ_LOOP
endtask: BODY
endclass: SEQUENCE
```

- Loop puts TRIALS transactions into mailbox SEQ2DRV.
- Each transaction is identified by its incremented TAG.
- Method .randomize() will only be called from this task.

A Simulated Sequence

Transcript
for Eight
Iterations

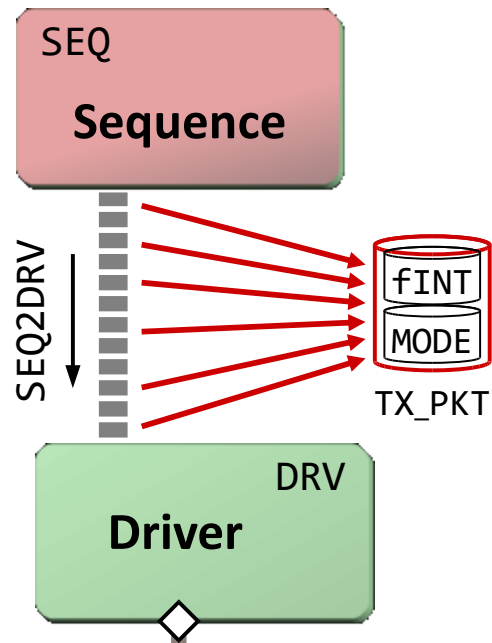
Zero Time

```
1.500 ms: Launch SEQ.BODY...
1.500 ms: # 1. M4_40-120x1 at 21 kHz
1.500 ms: # 2. M0_40-060x2 at 120 kHz
1.500 ms: # 3. M7_20-080x1 at 19 kHz
1.500 ms: # 4. M6_20-120x1 at 117 kHz
1.500 ms: # 5. M3_20-040x2 at 103 kHz
1.500 ms: # 6. M1_40-040x2 at 91 kHz
1.500 ms: # 7. M2_20-060x2 at 32 kHz
1.500 ms: # 8. M5_40-080x1 at 21 kHz
. . . . .
```

Same
*f*INT,
Distinct
MODE

- Series of packets transmitted in **zero time** at 1.50 ms.
- Left up to driver, **exactly when** to apply the stimulus.
- Task writes out real time, tag, mode, and frequency.
- As expected, modes **don't repeat**—but frequency can.

Exercise #2: Faulty Mailbox



Buggy
Task

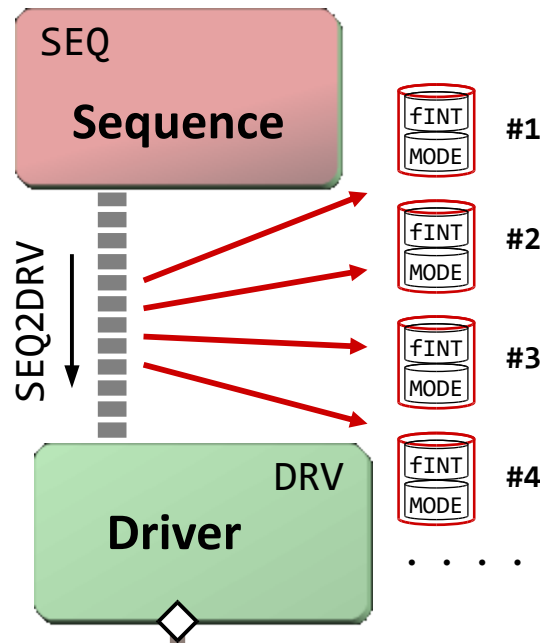
```
TX_PKT = new();
for(«Loop for TRIALS»)
begin:SEQ_LOOP
    TX_PKT.TAG = I;
    TX_PKT.randomize();
    SEQ2DRV.put(TX_PKT);
    $write(«packet data»);
end: SEQ_LOOP
```

Reference:

SystemVerilog for Verification (Springer: 2012),
Chris Spear & Greg Tumbush, Sample 7.32.

- This sequence BODY() task has hard-to-find **logical bug**.
- Recall: What is **put** in a mailbox is **not** an actual object.
- It is only the **pointer** to the object, like packet TX_PKT.

Answer #2: Robust Mailbox



```
TX_PKT = new();
for(«Loop for TRIALS»)
begin:SEQ_LOOP
  TX_PKT = new();
  TX_PKT.TAG = I;
  TX_PKT.randomize();
  SEQ2DRV.put(TX_PKT);
  $write(«packet data»);
end: SEQ_LOOP
```

Debugged BODY Task

- Reusing **same packet** will queue up same pointer 16x.
- Driver will apply only the **last set** of random stimuli!
- Solution: Move the **new** constructor **inside** the loop.

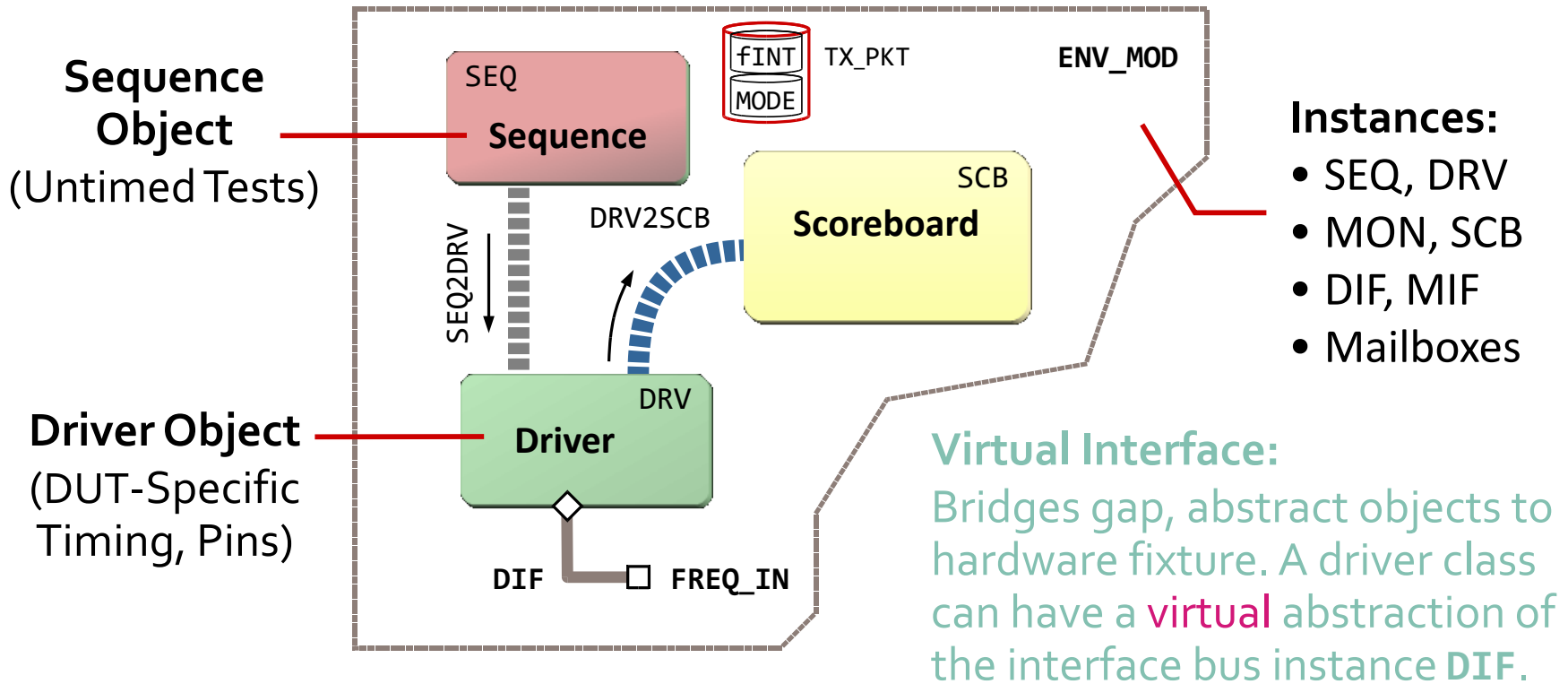
Summary: Sequence Object

- A. Can add one or more SEQUENCE components to an analog OOP testbench, to conform more to UVM.
- B. Optionally, parameterize the SEQUENCE class to control number of TRIALS, or other testing details.
- C. Sequence BODY() task creates the random stimuli, in untimed format, to be applied to DUT by a driver.
- D. An OOP testbench is quite adequate for stand-alone analog IP. Migrate to UVM for large, complex SOC.

§4: Driver and Monitor

- Driver Architecture
- Virtual Interface Explained
- Driver APPLY_SINE() Task
- Monitor SAMPLE_PPA() Task

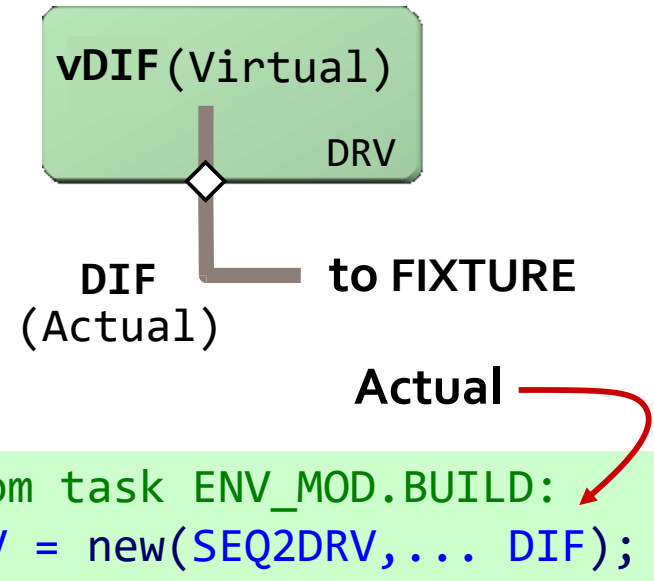
Driver Architecture



- In Part 1, the driver object **created** the random stimuli.
- Now, it merely receives packets from **sequence** object.
- Extracts fields from packet, then **applies them** to DUT.

Virtual Interface Explained

```
class DRIVER #(int TRIALS = 16);
    . . .
    //Interface to fixture port:
    virtual BAND_IF vDIF; //Null.
    //Call from ENV_MOD task BUILD:
    function new(
        mailbox SEQ2DRV_arg, . . .
        virtual BAND_IF DIF_arg
    ); . . .
        vDIF = DIF_arg; //Virtual.
    endfunction: new
    . . .
endclass: DRIVER
```



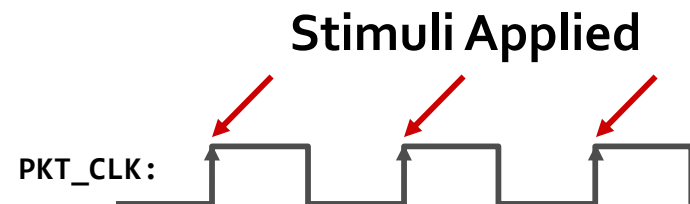
- A class **cannot** instantiate a hardware **interface** bus.
- Instead it declares a **virtual abstraction vDIF** of bus **BAND_IF**.
- When **DRV** is built, **actual DIF** is passed to **virtual vDIF**.

Driver APPLY_SINE() Task

```
//Call from ENV_MOD test suite:
task APPLY_SINE();
  for(«Loop for TRIALS»)
  begin:DRIVE_LOOP
    TX_PKT = new();
    SEQ2DRV.get(TX_PKT);
    //Apply stimuli to DIF:
    @(posedge vDIF.PKT_CLK);
    vDIF.TAG    = TX_PKT.TAG;
    vDIF.MODE   = TX_PKT.MODE;
    vDIF.fREAL  = TX_PKT.fREAL;
```

Virtual Interface

```
//Send packet to SCB:
  DRV2SCB.put(TX_PKT);
  $write(«packet data»);
end: DRIVE_LOOP
endtask: APPLY_SINE
```



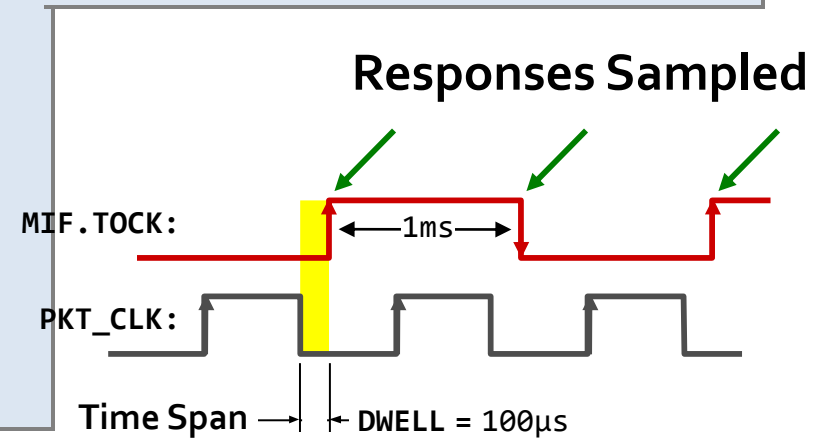
- Driver task can now refer to virtual **vDIF**, and its signals.
- Applies sequence **packet fields** to appropriate signals.
- Task is aware of DUT-specific **timing** and signal names.

Monitor SAMPLE_PPA() Task

```
task SAMPLE_PPA();
  @(posedge MIF.PKT_CLK);
  for(«Loop for TRIALS»)
  begin:SAMPLE_LOOP
    RX_PKT = new();
    //Sample just after edges:
    @(vMIF.TOCK) #(tSAMPLE); //µs.

    . . . . .
    RX_PKT.PPA_IN  = vMIF.PPA_IN;
    RX_PKT.PPA_OUT = vMIF.PPA_OUT;
    //Collect coverage metrics:
    CVG.sample;
```

```
//Send packet to SCB:
  MON2SCB.put(RX_PKT);
  $write(«packet data»);
end:  SAMPLE_LOOP
endtask:  SAMPLE_PPA
```



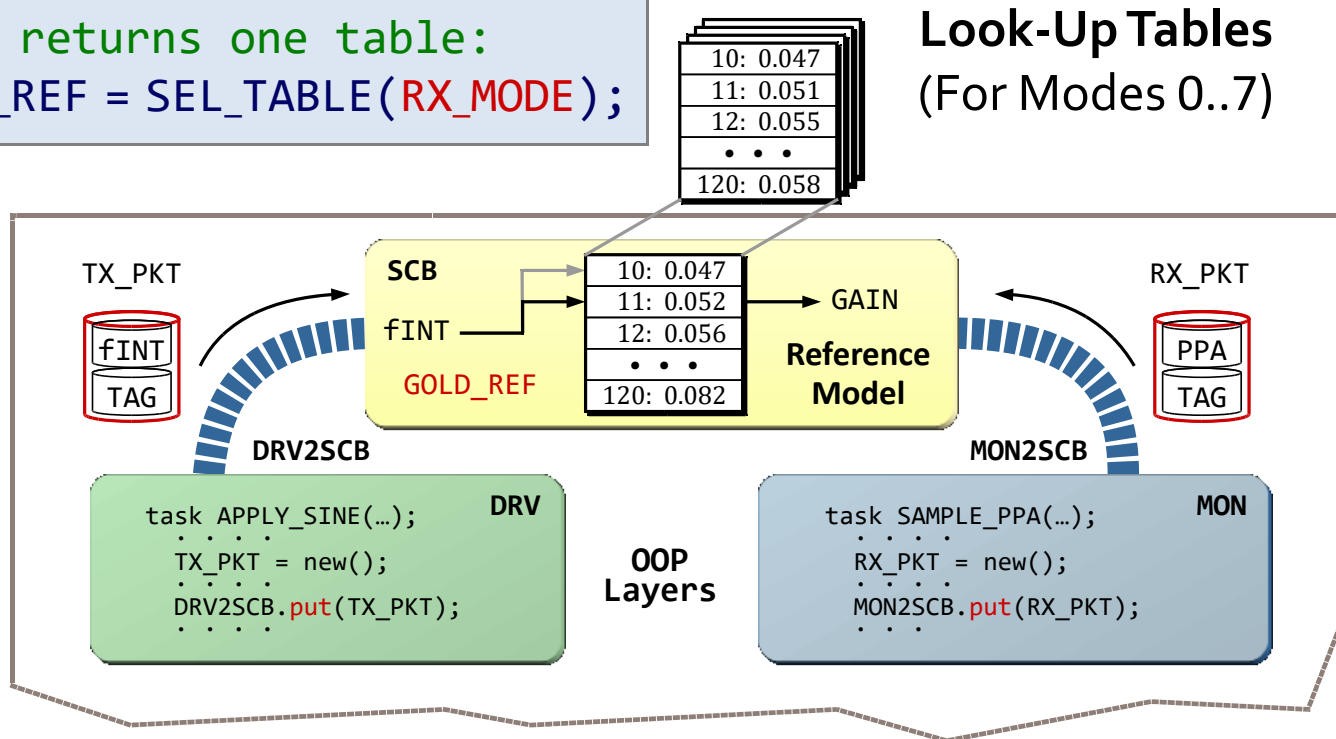
- **Monitor** organized like the driver, with virtual **vMIF** bus.
- Samples **vMIF** signals, **reassembling** them into packets.
- Task is aware of hardware **timing**; allows settling time.

§5: The Scoreboard

- Scoreboard Architecture
- Table Selection Code
- Sample Look-Up Table
- SPICE-Generated Tables
- Array Look-Up Function
- Scoreboard Transcript

Scoreboard Architecture

```
//Call returns one table:  
GOLD_REF = SEL_TABLE(RX_MODE);
```



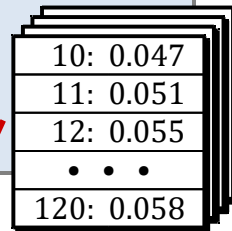
- Scoreboard must compare **actual** gain versus **expected**.
- Expected response for any mode is in a **look-up table**.
- Table is first **selected** by calling `SEL_TABLE(RX_MODE)`.

Table Selection Code

```
//Look-up table type:
typedef
//Volts      kHz:
  real  AA_t [int];

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

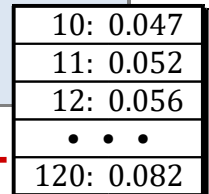
Array of Tables



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

```
function AA_t SEL_TABLE(MODE);
  case (MODE) inside
    \M0_40-060x2 :
      return(GOLD_REFS[0]);
    \M1_40-040x2 :
      return(GOLD_REFS[1]);
    . . . . .
    \M7_20-080x1 :
      return(GOLD_REFS[7]);
  endcase
endfunction: SEL_TABLE
```

Look-Up Table 0



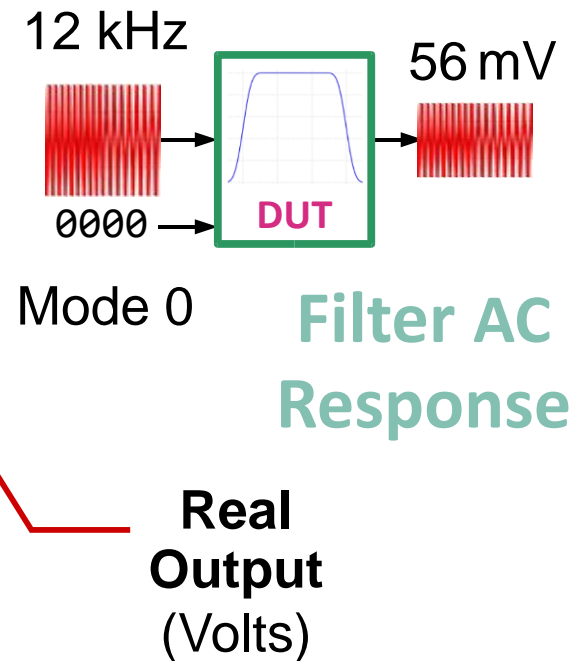
10: 0.047
11: 0.052
12: 0.056
. . .
120: 0.082

- From monitor's RX_PKT, the **current mode** is extracted.
- A **case** statement returns **look-up table** for that mode.
- Now, AA_t variable **GOLD_REF** holds the correct table.

Sample Look-Up Table

Integer
Index
(kHz)

```
/* Associative array:
 * Format: '{fINT: Vout, ...}'
 */
real GOLD_REF_0[int] = '{
  int'( 10.00000): 0.047_9123,
  int'( 11.00000): 0.052_2140,
  int'( 12.00000): 0.056_3879,
  . . . . .
  int'(119.00000): 0.083_2204,
  int'(120.00000): 0.082_7318,
  default: 0.000_0000
};
```



- An **associative array** is convenient for a look-up table.
- Allows even **noncontiguous** frequency-response data.
- For 12 kHz input, **GOLD_REF_0** expects ~56 mV output.

SPICE-Generated Tables

```
.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

awk
Script

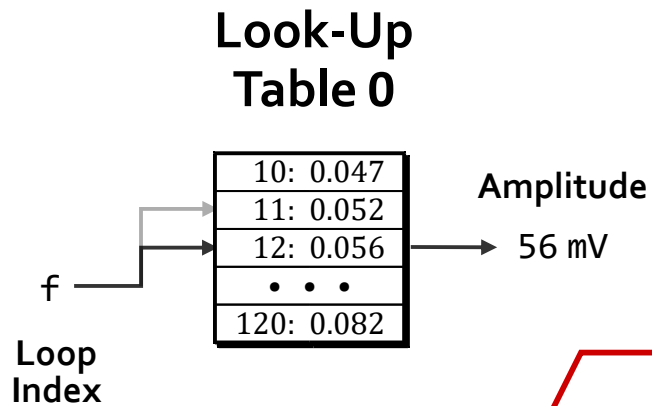
```
//Associative array:
```

```
//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
};
```

- 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 a SystemVerilog array.

Array Look-Up Function



Scan for least delta:

$$fDELTA = |fINT - f|$$

```
function real GOLD_VOUT(int fINT);
  int fDELTA, fLAST, fBEST; . . .
  //Initialize to first difference:
  fLAST = ABS(fINT - fMIN);
  foreach (GOLD_REF[f])
  begin:SCAN
    fDELTA = ABS(fINT - f);
    if (fDELTA <= fLAST) fBEST = f;
    fLAST = fDELTA;
  end:  SCAN
  return (GOLD_REF[fBEST]);
endfunction: GOLD_VOUT
```

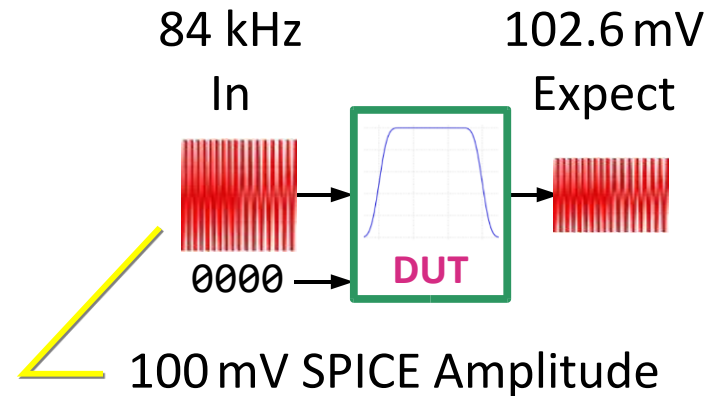
- With selected table in GOLD_REF, we **visit** every index.
- Scan for the least delta, remembering the **best** index.
- Return **GOLD_REF[fBEST]** as the expected amplitude.

Scoreboard Transcript

```

/* Compute expected filter gain from
 * amplitude returned by GOLD_VOUT():
 */
GOLD_REF = SEL_TABLE(RX_MODE);
gEXPECT  = GOLD_VOUT(fINT)/0.100;
ERR_ABS  = gACTUAL - gEXPECT;

```



From
XMODEL
Transcript

```

<Score: TX(15)==RX(15)> M0_40-060x2 @84 kHz
gACTUAL: PPA_OUT/IN = 0.20859447/0.200 = 1.04297233
gEXPECT: SCB.GOLD_VOUT( 84 kHz)/0.100 = 1.02609800
|ERROR|: gACTUAL - gEXPECT = 0.01687433

```

- Then **expected gain** = table look-up ÷ SPICE amplitude.
- Works even for a noncontiguous frequency response.
- Across packets, worst-case error only **1 or 2%** of SPICE.

§6: Bypass/Power-Down

- Four Descriptive States
- Extended Test Sequence
- Driving the Control Bits
- Summary: Extended Test

Four Descriptive States

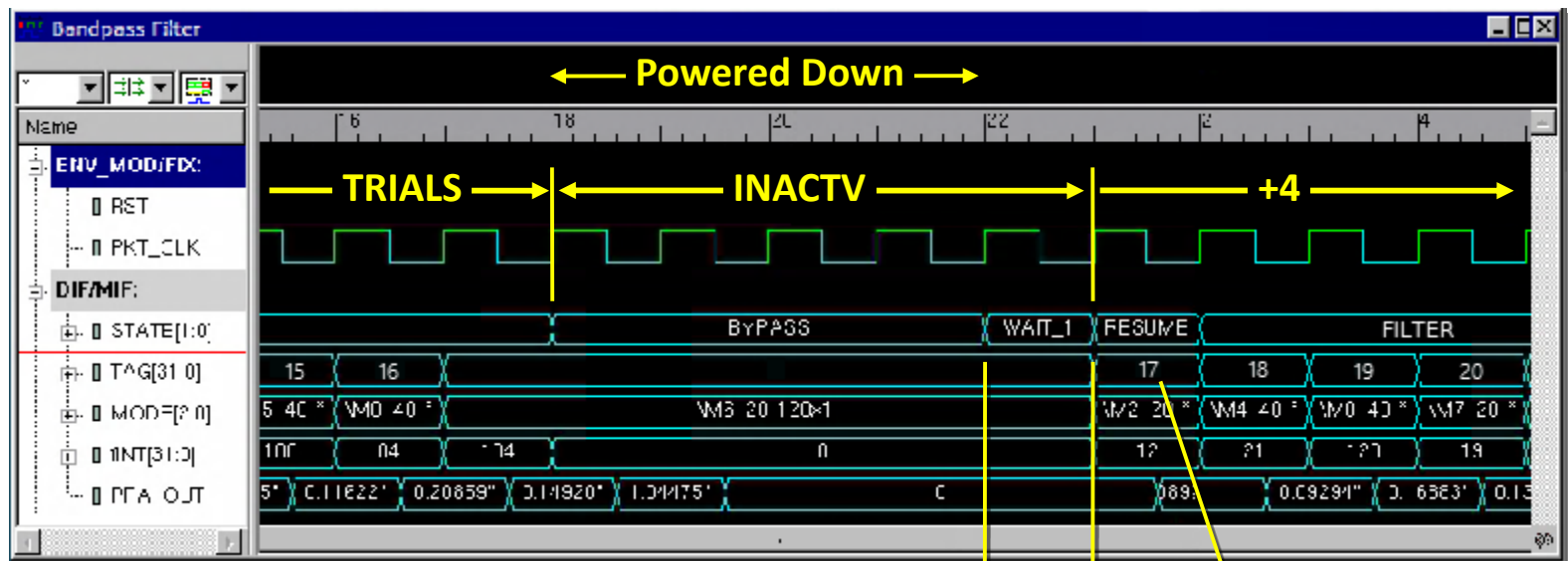
```
//Define four DUT control states:
typedef
    enum bit [1:0] {
        FILTER = 2'b00, //Normal operation.
        BYPASS = 2'b11, //Bypass/power-down.
        WAIT_1 = 2'b10, //One power-up cycle.
        RESUME = 2'b01 //Resume operation.
    } STATE_t;
```

State Names
Visible in
Wave View

Imported from GOLD_PKG where needed.

- Extend **basic** test sequence—add **bypass** and **recovery**.
- First do TRIALS iterations in the normal **filtering** mode.
- Then **bypass** DUT for a few cycles, and power up again.
- **Resume** filtering a few more cycles to check recovery.

Extended Test Sequence



One-cycle WAIT_1 state allows for Vdd rise time.

Use TAG saved during bypass.

- Arbitrarily set INACTV to 5 (4 **BYPASS** + **WAIT_1**) cycles.
- Followed by a **RESUME** cycle, which preserves last TAG.
- Wrap up with 3 **FILTER** cycles, which add to coverage.

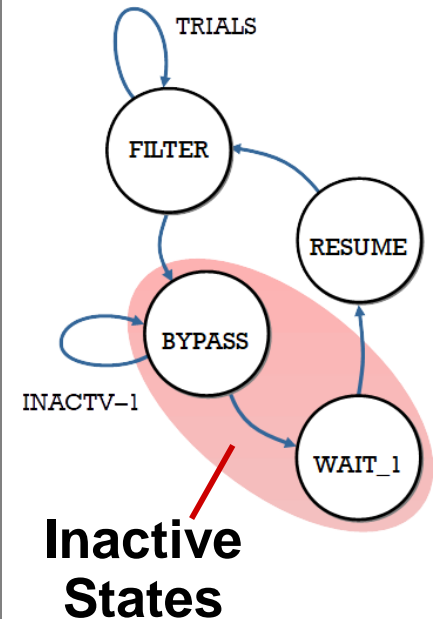
Driving the Control Bits

**Bypass
Mode**

**Powering
Up Again**

```
//Drive CTRL from MODE and STATE:
always @(posedge FREQ_IN.PKT_CLK)
  case (STATE)
    BYPASS: begin
      CTRL[3]    = 1'b1;
      CTRL[2:0] = «default»; end
    WAIT_1: begin
      CTRL[3]    = 1'b0;
      CTRL[2:0] = «default»; end
    «RESUME case item»
    FILTER: begin
      CTRL[3]    = 1'b0;
      CTRL[2:0] = MODE; end
  endcase
```

**Fixture
Code**



- Organizing the code **by state** made the details clearer.
- The **CTRL** bits drive DUT inputs {ct1_byp,...,ct1_c2}.

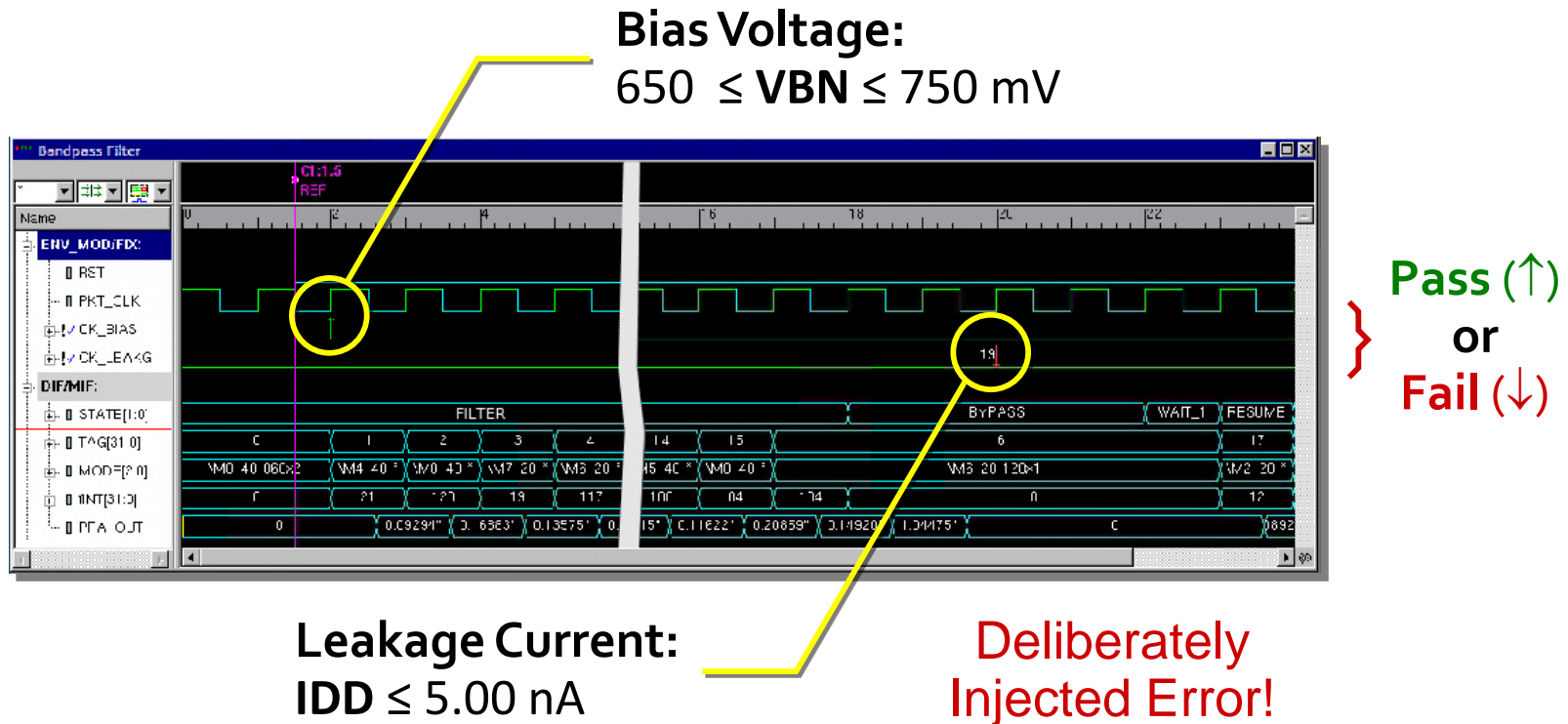
Summary: Extended Test

- A. Bypass and recovery modes were **not randomized**, but coded as a **directed** test case, using convenient parameters TRIALS and INACTV (default: 5 cycles).
- B. Four descriptive **states** defined, to better organize code as DUT was bypassed, then powered back up.
- C. During bypass and recovery, collection of coverage metrics must be **suspended** using **iff** clause [§8].
- D. Critical design properties can still be checked during the bypass cycles, using **analog assertions** [§7].

§7: Analog Assertions

- Viewing Analog Assertions
- An Assertion Subcircuit
- Analog Assertion Plan
- Asserting Voltage, Current
- Using Assertion Defaults

Viewing Analog Assertions



- SystemVerilog **assertion** (SVA) syntax is rich, growing.
- Assertions **complement** any self-checking testbench.
- Why not use **analog** assertions to verify key properties?

— 56



- XMODEL**

Analog Assertion Plan

Analog/Mixed-Signal Assertions for Filter Properties				
Assertion Label	Variable Checked	Antecedent Condition	Property Expression	XMODEL Primitives
Voltage { CK_BIAS	Vbn (nMOS Bias)	RST deasserted	$V_{bn} = 700 \pm 50 \text{ mV}$	xreal_to_real
Current { CK_LEAKG	Idd (Leakage)	STATE==BYPASS	$I_{dd}(\text{max}) \leq 5 \text{ nA}$	iprobe, meas_max

Optional Trigger Condition:

- Start checks in **same** cycle: $\mid \rightarrow$
- Start checks in **next** cycle: $\mid \Rightarrow$

Info: **Scientific Analog** website, **xmodel** button. See FEATURE 2: scianalog.com/xmodel

- Highly **localized**; a failure leads right to hardware bug.
- Catches predictable issues, such as **polarity mismatch**.
- Document DUT-specific assumptions like **invalid state**.
- Access **purely analog** design features, like a rise-time.

Asserting a Bias Voltage

```
//Check nMOS bias after a reset:
property BIASING_pro;
  @(posedge FREQ_IN.PKT_CLK)
    $fell(FREQ_IN.RST) |->
      (VBN >= 0.650) && (VBN <= 0.750);
endproperty: BIASING_pro

CK_BIAS: assert property (BIASING_pro)
  «Report pass; else report failure.»
```

LRM Clause 16.6:

Boolean Expressions

Subexpression can be of **any type**, as long as the overall expression still evaluates true or false.

SVA Code in Fixture

- First define the **property** BIASING_PRO, then assert it.
- Checking is only **triggered** after RST falls to inactive.
- Thereafter, **checking** is at every posedge of PKT_CLK.

Asserting Leakage Current

```
//Check leakage current in bypass:
property LEAKAGE_pro;
  @(posedge FREQ_IN.PKT_CLK)
    $rose(STATE == BYPASS) | =>
      IDD < 5e-9; //Inject a failure.
endproperty: LEAKAGE_pro

CK_LEAKG: assert property (LEAKAGE_pro)
  «Report pass; else report failure.»
```

Injected Failure:

A reported failure, or a **red arrow** in Wave View, pinpoints exact locale of the underlying bug.

SVA Code in Fixture

```
//Example of a sequence expression:
(IDD < 5e-9) throughout (STATE==BYPASS)[*4];
```

- Can defer checking to **next** cycle to allow settling time.
- Seamlessly integrates with a **full range** of SVA syntax.
- Define a **chain of events** using the **sequence** keyword.

Using Assertion Defaults

Empty Clocking Block CB

```
//Default clocking for assertions:
clocking CB
  @(posedge FREQ_IN.PKT_CLK);
endclocking: CB

default clocking CB;

//Disable checking during a reset:
default disable iff (FREQ_IN.RST);

//Check leakage current in bypass:
property LEAKAGE_pro;
  @(posedge FREQ_IN.PKT_CLK)
    $rose(STATE == BYPASS) | =>
      . . . .
endproperty: LEAKAGE_pro
```

Default Clock for Assertions Within Scope

Omit Clock in Property

Prevents Checking During an Asynch Reset

SVA Code in Fixture

- Assertions can **share** default clock edge, and a disable.
- More concise, when you're writing **multiple** properties.

§8: Functional Coverage

- A Basic Coverage Group
- Frequency Coverage Map
- Enhanced Coverage Group
- Functional Coverage Stats
- Cross-Coverage Map
- Tactic: In-Line Constraint
- Brute-Force Approach
- Summary: Coverage

A Basic Coverage Group

Embedded
in Monitor
Class

Each Bin Must
Be **Populated**
During a Run



```
//Coverage of frequency only:
covergroup CVG;
//Applied input-frequency bins:
  FREQ_cvg: coverpoint RX_PKT.fINT
  {
    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
```

Direct Access
to Monitor
Properties

Partition
Frequency
Range of
10–120 kHz
into **6 Bins**

```
//Collect metrics:
  CVG.sample;
```

- In Part 1, only the analog **frequency** was randomized.
- A **coverpoint** tells VCS to compile coverage statistics.

Frequency Coverage Map

Frequency Bin
(Histogram Bar)

Metrics:

Total trials = 20

Filled Bins = 6

Total Bins = 6

Coverage = 100%

<i>f</i>	<i>n</i>
B10	1
B20	5
B40	4
B60	2
B80	3
B100	5

```
unix> ${VCS_HOME}/bin/urg -dir simv.vdb
```

Summary for Variable FREQ_CVG

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined	6	0	6	100.00

User Defined Bins for FREQ_CVG

NAME	COUNT	AT LEAST	TEST	COUNT
B100	5	1	T1	5
B80	3	1	T1	3
B60	2	1	T1	2
B40	4	1	T1	4
B20	5	1	T1	5
B10	1	1	T1	1

VCS
urg



- This simulation ran for 20 trials, easily achieving 100%.
- Every bin, shaded in green, has population of **at least** 1.
- Data was extracted from a VCS **unified coverage** report.

Enhanced Coverage Group (1/2)

Name of
Cover Point
Used in urg

One Group
Can Specify
Multiple
Cover Points

```
//Cross-coverage of mode, frequency:
covergroup CVG;
  FREQ_cv: coverpoint RX_PKT.fINT
  iff (
    RX_PKT.STATE == FILTER ||
    RX_PKT.STATE == RESUME )
  {
    bins B10  = {[ 10: 19]};
    bins B20  = {[ 20: 39]};
    . . . .
    bins B80  = {[ 80: 99]};
    bins B100 = {[100:120]};
  } . . . .
```

Don't Sample
in **Bypass** or
Wait States

- In Part 2, we also randomized the digital **mode** word.
- Let's add a cover point for **MODE** to the **same group**.

Enhanced Coverage Group (2/2)

Direct Access
to Monitor
Properties

```

    . . . .
    //Applied input-mode values:
    MODE_cvg: coverpoint RX_PKT.MODE
    iff (
        RX_PKT.STATE == FILTER ||
        RX_PKT.STATE == RESUME
    );
    //Cross-coverage:  MODE x fINT
    CROSS_cvg:
        cross MODE_cvg, FREQ_cvg;
    endgroup: CVG
  
```

Sample Only
in **Normal**
Operation

Cartesian
Cross-Product
(All Possible
Combinations)

List of
Cover Point
Names

- Covering all 8 modes is easy. But what about $f \times \mathbf{MODE}$?
- Keyword **cross** tells VCS to compile a 2-D histogram.
- Now there are $6 \times 8 = 48$ histogram bars, or bins, to fill.

Functional Coverage Stats

```
unix> ${VCS_HOME}/bin/urg -dir simv.vdb
```

100%
Coverage
of Both
Random
Variables

Name of
Cross Point

Variables for Group MON_PKG::MONITOR#(40,5)::CVG						
VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	
FREQ_cvg	6	0	6	100.00	100	
MODE_cvg	8	0	8	100.00	100	

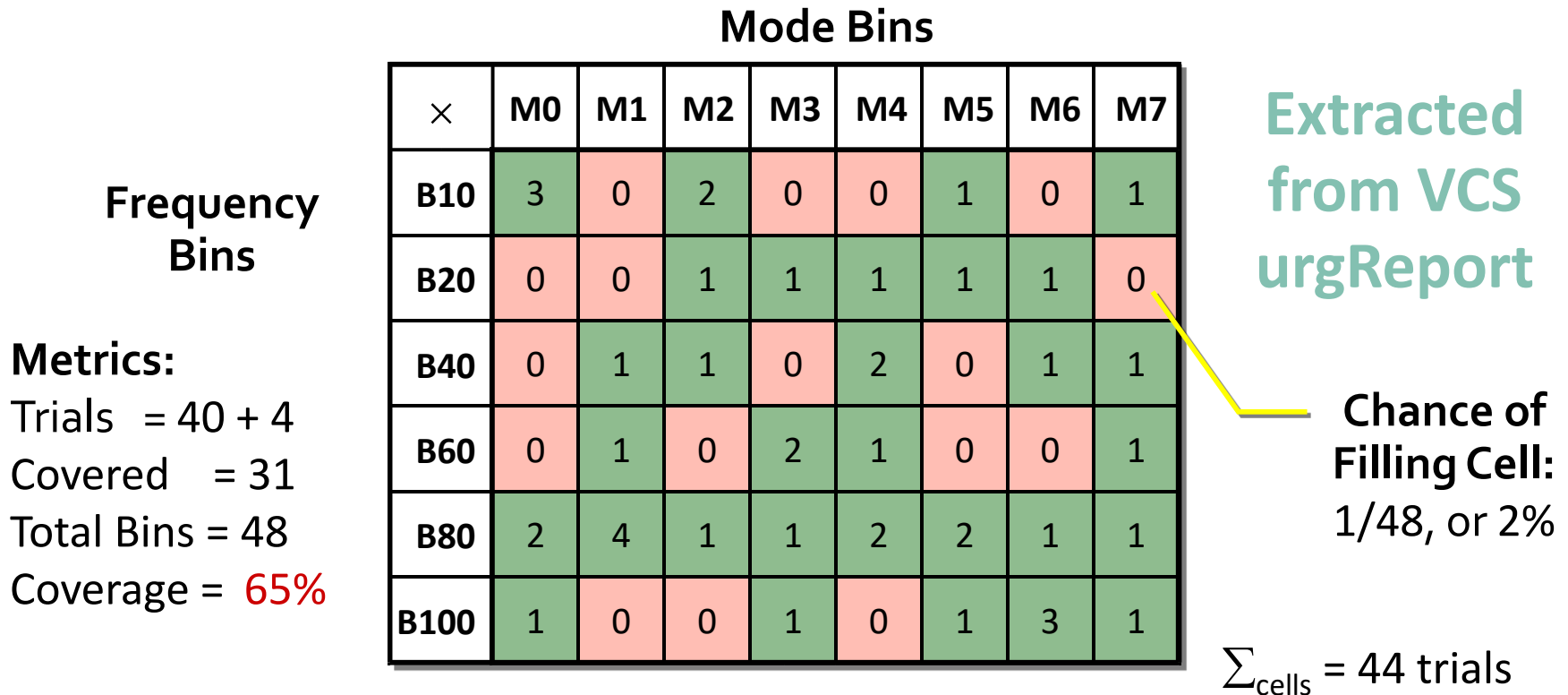
Crosses for Group MON_PKG::MONITOR#(40,5)::CVG						
CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
CROSS_cvg	48	17	31	64.58	100	1

urgReport
(HTML View)

Why is Cross
Coverage Low?

- Individual coverage of frequency and of mode is 100%.
- But cross coverage is only 65%, after a run of 40 trials.
- Recall that default **rand** yields a uniform distribution.

Cross-Coverage Map

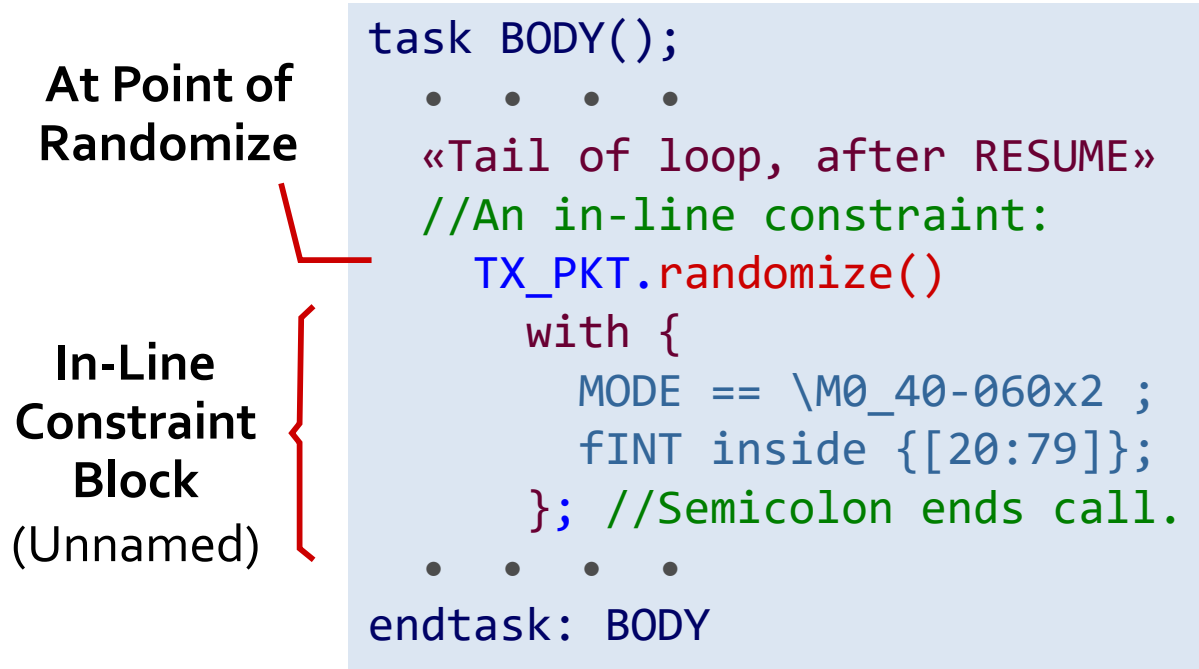


Metrics:

Trials = 40 + 4
 Covered = 31
 Total Bins = 48
 Coverage = 65%

- A **low probability** to hit any one cell— e.g. (B20, M7).
- Elegant solution: try various SystemVerilog **tactics**.
- **Brute-force** solution: keep simulating for ~200 trials.

Tactic: In-Line Constraint (1/2)



×	M0
B10	3
B20	0
B40	0
B60	0

Empty Bins



- Can add extra constraints **in line** with .randomize call.
- Tactic attempts to fill up three specific bins still **empty**.
- A valid approach—but in this case causes fatal **conflict**.

Tactic: In-Line Constraint (2/2)

RUN-TIME ERROR:
 Solver failed when solving this set of constraints:

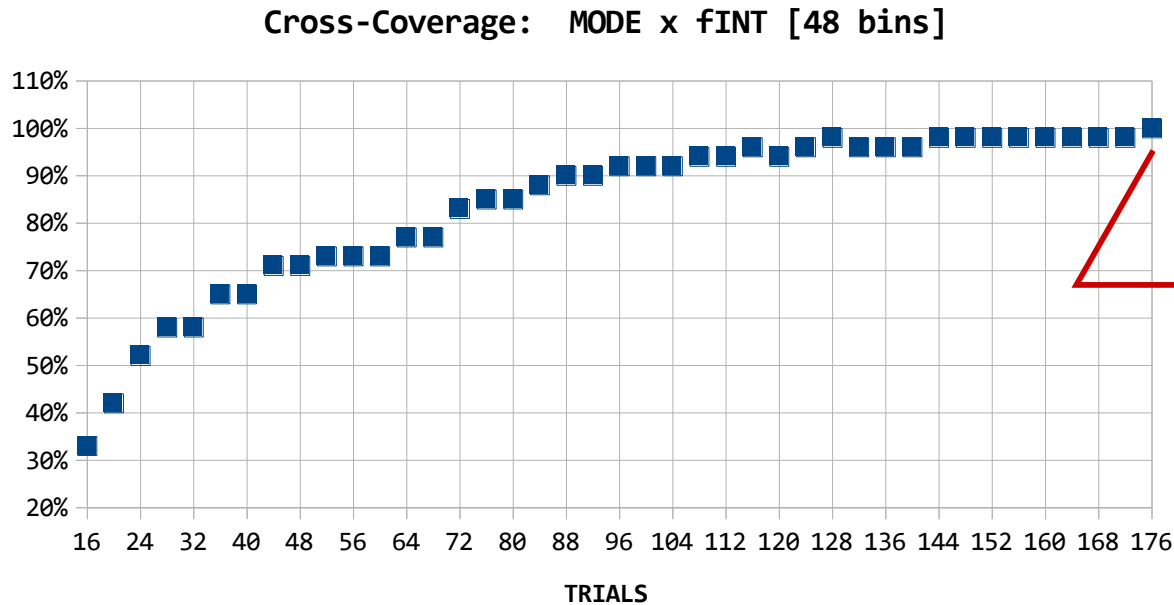
Packet Object {
 MODE_t USED[1] = '{\M0_40-060x2 }';
 constraint XCLUDE_con {
 !(MODE inside {USED});
 }
 }
Sequence Object — constraint WITH_CONSTRAINT { //Unnamed constraint.
 MODE == \M0_40-060x2 ;
 }

Reference: Dave Rich (Siemens/Mentor),
SystemVerilog Constraints: What You Forgot in School

- Solver considers all constraints **at once**; no prioritizing.
- Can **conflict**—especially if specified in multiple objects.
- Mode **M0** was in **USED** queue when we tried to **reuse** it.
- Solver issued this **run-time** error, for the packet #178.

Brute-Force Approach

Random
Seed:
3947



**100% Cross
Coverage
After 176
Trials**

STATISTICS

Total packets run: 180 tested; 180 matching; 0 mismatch.
Worst gain error: 0.01727394

*** XMODEL/VCS simulation ***
started at: 23:34:41
ended at : 23:35:22
total time: 00:00.41

**Simulation
Wall Time:
~41 sec**

Summary: Functional Coverage

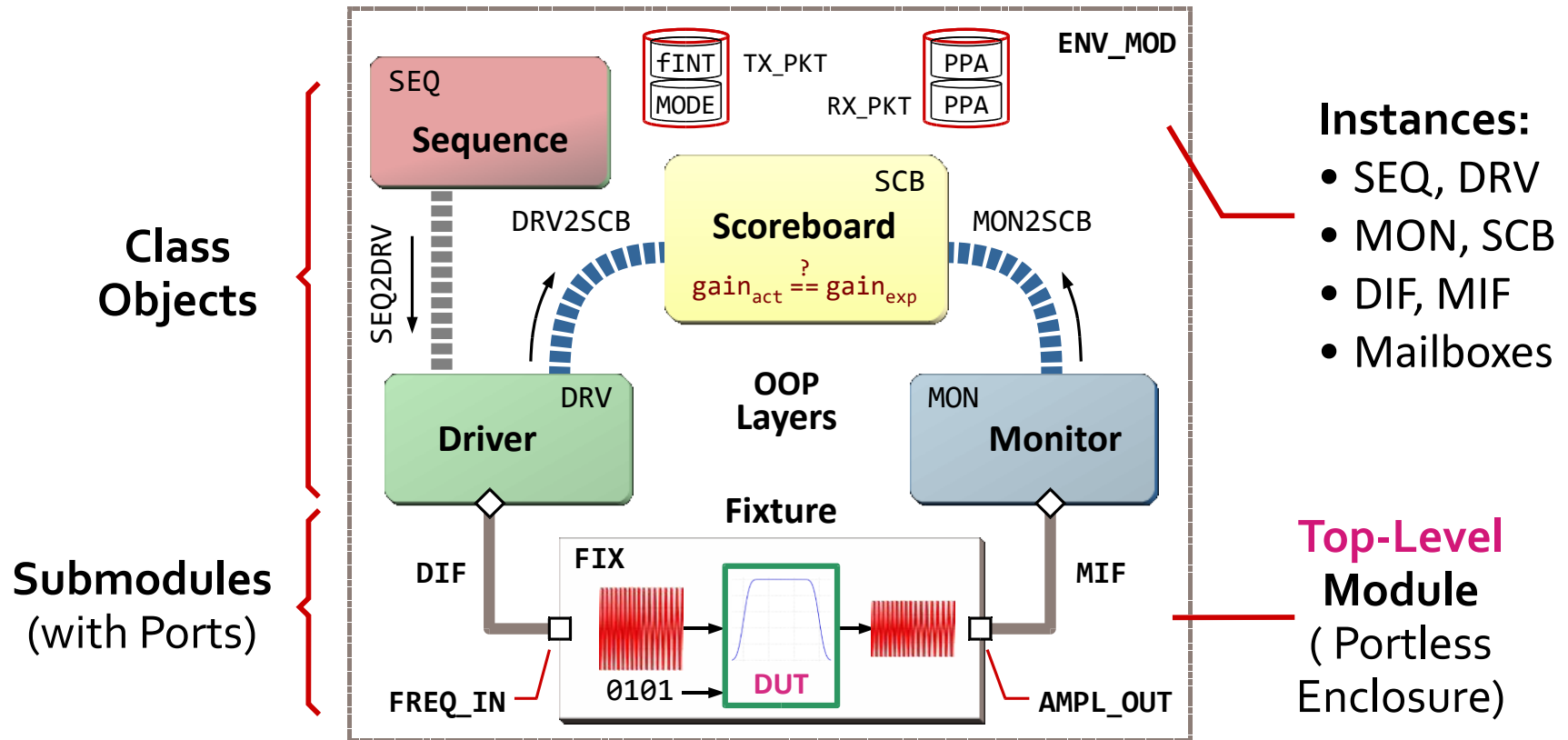
- A. Verification teams can ensure complete coverage of **key features** of a digitally-controlled analog DUT.
- B. The XMODEL verification flow enables seamless integration of **functional coverage** constructs.
- C. Compatible with all of SystemVerilog's coverage syntax: **coverpoint, cross, sample, bins, iff, with....**
- D. **Speed** of XMODEL simulations makes brute-force iteration— e.g. 10^3 analog DUT trials—practical.

scientific
analog

§9: Environment Module

- Environment Architecture
- Top-Level Test Suite
- Sample Transcript

Environment Architecture



- In UVM, top-level environment is a **class**, like **uvm_env**.
- For simplicity, we've used a top-level **module ENV_MOD**.

Top-Level Test Suite

```

initial
begin:SUITE
//Build phase:
  BUILD();
//Run phase:
  wait(!RST);
  SEQ.BODY();
  fork
    DRV.APPLY_SINE();
    MON.SAMPLE_PPA();
  join

```

UVM-Like
Phases

```

//Check phase:
  SCB.SCORE_PACKET();
  @(posedge PKT_CLK) $finish;
end: SUITE

```

```

XMODEL Manifest File (man.f):
--simtime 200ms
--sim-option -assert quiet --
--sim-option +ntb_random_seed=3947 --
--logfile RUN-3947.log

```

- Test suite first **constructs** all OOP components, at 0 ms.
- After reset, generates **sequence** by calling task BODY().
- Applies DUT inputs; **concurrently** samples its outputs.
- Finally, it **scores** the packets sent from driver, monitor.

Sample Transcript

Random
Seed:
3947

```
186.605 ms: <Score: TX(179)==RX(179)> M6_20-120x1, 34 kHz
gACTUAL: PPA_OUT/IN = 0.16768778/0.200 = 0.83843891
gEXPECT: SCB.GOLD_VOUT( 34 kHz)/0.100 = 0.83368500
|ERROR|: gACTUAL - gEXPECT (absolute) = 0.00475391
```

```
186.605 ms: <Score: TX(180)==RX(180)> M3_20-040x2, 64 kHz
gACTUAL: PPA_OUT/IN = 0.20078843/0.200 = 1.00394215
gEXPECT: SCB.GOLD_VOUT( 64 kHz)/0.100 = 0.99510200
|ERROR|: gACTUAL - gEXPECT (absolute) = 0.00884015
```

```
186.605 ms: Finish SCORE_PACKET...
```

```
187.000 ms: STATISTICS
```

```
Total packets run: 180 tested; 180 matching; 0 mismatch.
```

```
Worst gain error: 0.01727394
```

- Transcript shows scoring for last two of **180 packets**.
- Just alter **random seed** for different sequence of data.
- Thus, you **reuse** same OOP code for dozens of tests.

§10: Guidelines

- Guidelines 1–4
- Guidelines 5–9
- Guidelines 10–12

Note:

For easier reference, the relevant slide numbers have been indicated in boldface in square brackets—e.g., slide [50].

Guidelines 1–4

1. Develop a simplified DUT **macro model** to debug testbench in early OOP/UVM/DUT development.
2. Encapsulate XMODEL elements in a **fixture** module, including xrea1 signals; concurrent assertions **[28]**.
3. For maximum code reuse, put stimulus generation and randomization into an **untimed sequence**—analogous to class `uvm_sequence_item` **[32]**.
4. **Fork** the APPLY_SINE and MONITOR_PPA tasks, to measure PPA_OUT while fREAL is still applied **[74]**.

Guidelines 5–9

5. Use a static **queue** to get cyclic-random behavior, when limitations on **randc** are encountered [22].
6. Specify **random seed** on the command line [74].
7. Embed cover groups within a **monitor** class. Must still be **instantiated** to collect coverage data [62].
8. Avoid oversampling of coverage with **iff** clause—for example, while DUT is powered down [64].
9. Use **cross-coverage** of two (or more) random variables, to check on key corner cases [65].

Guidelines 10–12

10. Suppress excessive assertion reporting, using simulator-specific options [74].
11. Use default **disable** clause for assertions, under conditions like active asynchronous reset [60].
12. **Enumerate** DUT control variables or states, using **enum**. Coverage bins are automatically labeled with the enumerated names [50].

References:

C. Dančák, ResearchGate, *SystemVerilog OOP Testbench...Analog Filter*, Part 1 & 2.

www.researchgate.net/publication/

346061868_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_1

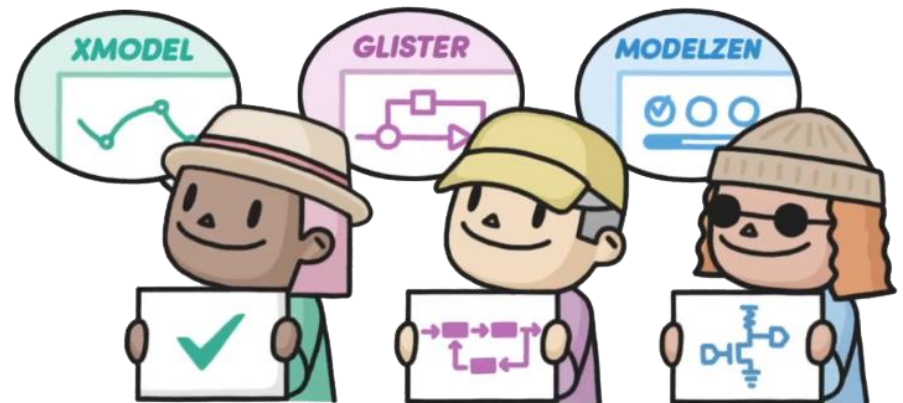
350412143_SystemVerilog_OOP_Testbench_for_Analog_Filter_A_Tutorial_Part_2

Final Summary

- This talk addressed how to write an OOP-style SystemVerilog testbench for a digitally-controlled analog DUT
- The testbench seamlessly integrates all language constructs of SystemVerilog — class components, interface bus, constrained randomizations, assertions, coverage, etc.
- XMODEL offers primitives for modeling analog circuits, generating stimuli, and checking results within SystemVerilog and provides 10~100x faster speed than RNV and SPICE-level accuracy

Resource List

- Watch again the pre-recorded videos of this webinar:
<https://youtu.be/gLfTYCLbj9M>
- Read tutorials and try with the examples:
<https://www.scianalog.com/webinar/20210625>
- Learn more about XMODEL:
<https://www.scianalog.com/xmodel>



Thank you!

