2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

Harnessing the Power of UVM for AMS Verification with *XMODEL*

Jaeha Kim and Charles Dančak
Scientific Analog, Inc.

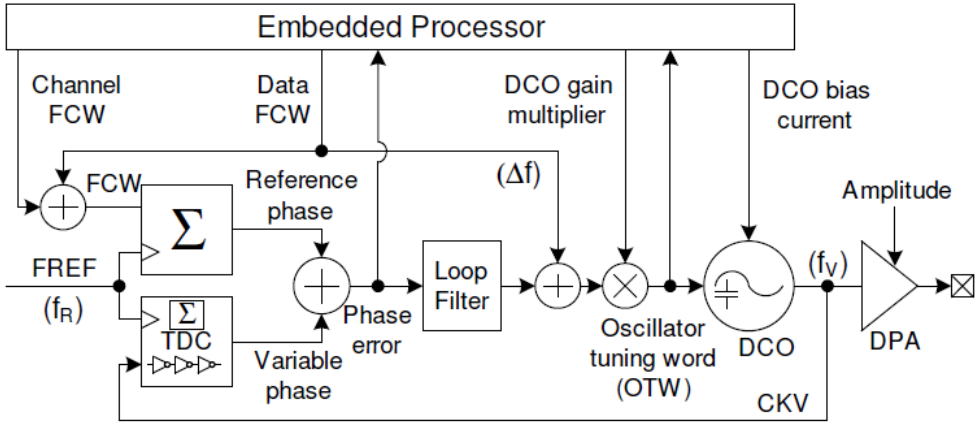
scientific analog

accellera SYSTEMS INITIATIVE

Why Verify Analog with SystemVerilog/UVM?

- ***UVM* (*Universal Verification Methodology*)** is a framework of building reusable & scalable testbenches for digital systems with standardized components

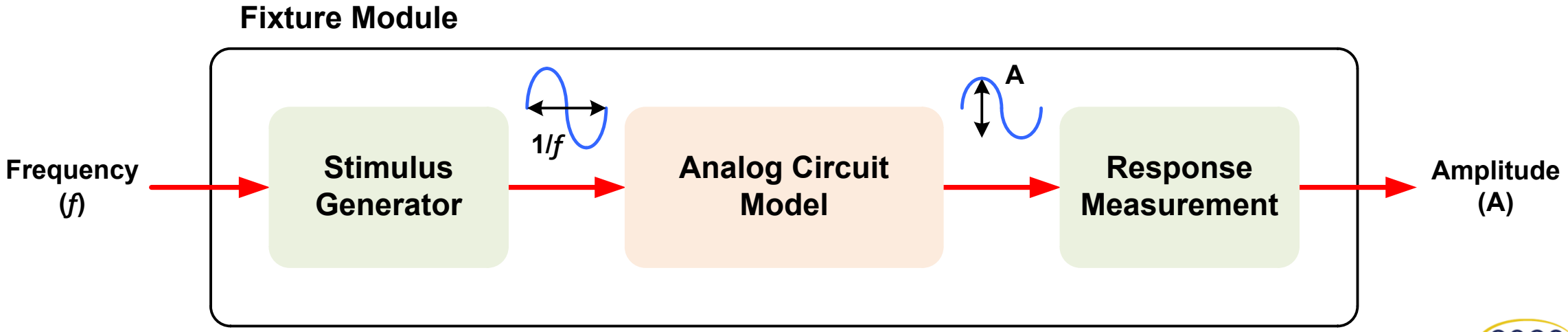
- Surge of ***analog circuits with digital programmability*** calls for UVM-like verification approaches extended to analog



[R. Staszewski, 2010]

Extending UVM to AMS Verification

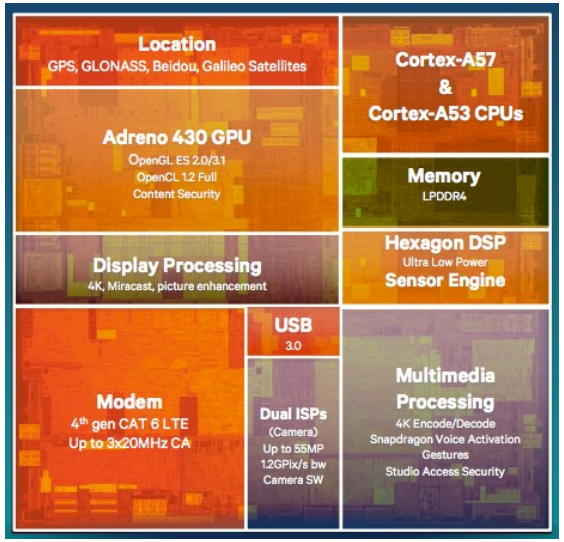
- UVM testbenches for AMS circuits can be built with standard components if we use a well-defined ***fixture module*** enclosing these elements:
  - AMS device under verification (DUV) modeled in SystemVerilog
  - Analog instrumentations for generating stimuli and measuring responses



Outline of This Tutorial

- In this tutorial, you'll learn how to write UVM testbenches for AMS circuits in a step-by-step manner:
  - Writing AMS instrumentations in SystemVerilog using *XMODEL* primitives
  - Auto-extracting SystemVerilog models from analog circuits with *MODELZEN*
  - Building a fixture module for an example programmable bandpass filter
  - Putting together the UVM components to compose a UVM testbench for AMS verification

SystemVerilog Testbench (UVM)



[Snapdragon 808/810, Qualcomm]

Instructors

Jaeha Kim

- Professor, Seoul National University, Korea
- CEO & Founder, Scientific Analog, Inc., CA
- MS & PhD in EE
- Served on TPCs of DAC, ICCAD, CICC & ISSCC
- Assoc. Editor of IEEE Journal of Solid-State Circuits

Charles Dančak

- Verification Instructor & Consultant, Betasoft Consulting, Inc., CA
- MS in EE and Physics
- Teaching SystemVerilog at UCSD Extension since 2007
- Author of DVCon US 2022 paper "UVM Testbench for AMS Verification"

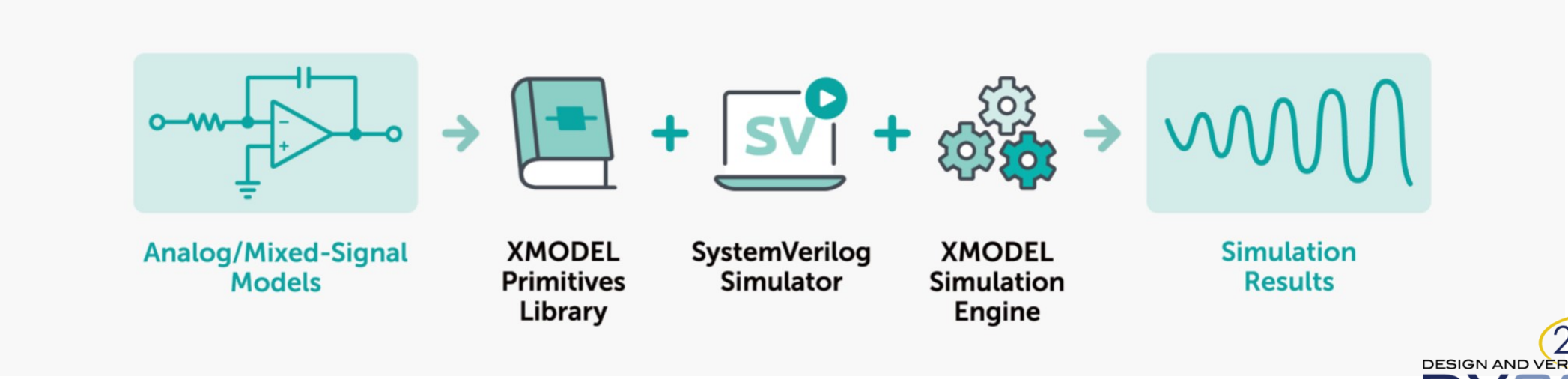
Part I. Building a Fixture Module for AMS Circuits

Jaeha Kim

scientific analog

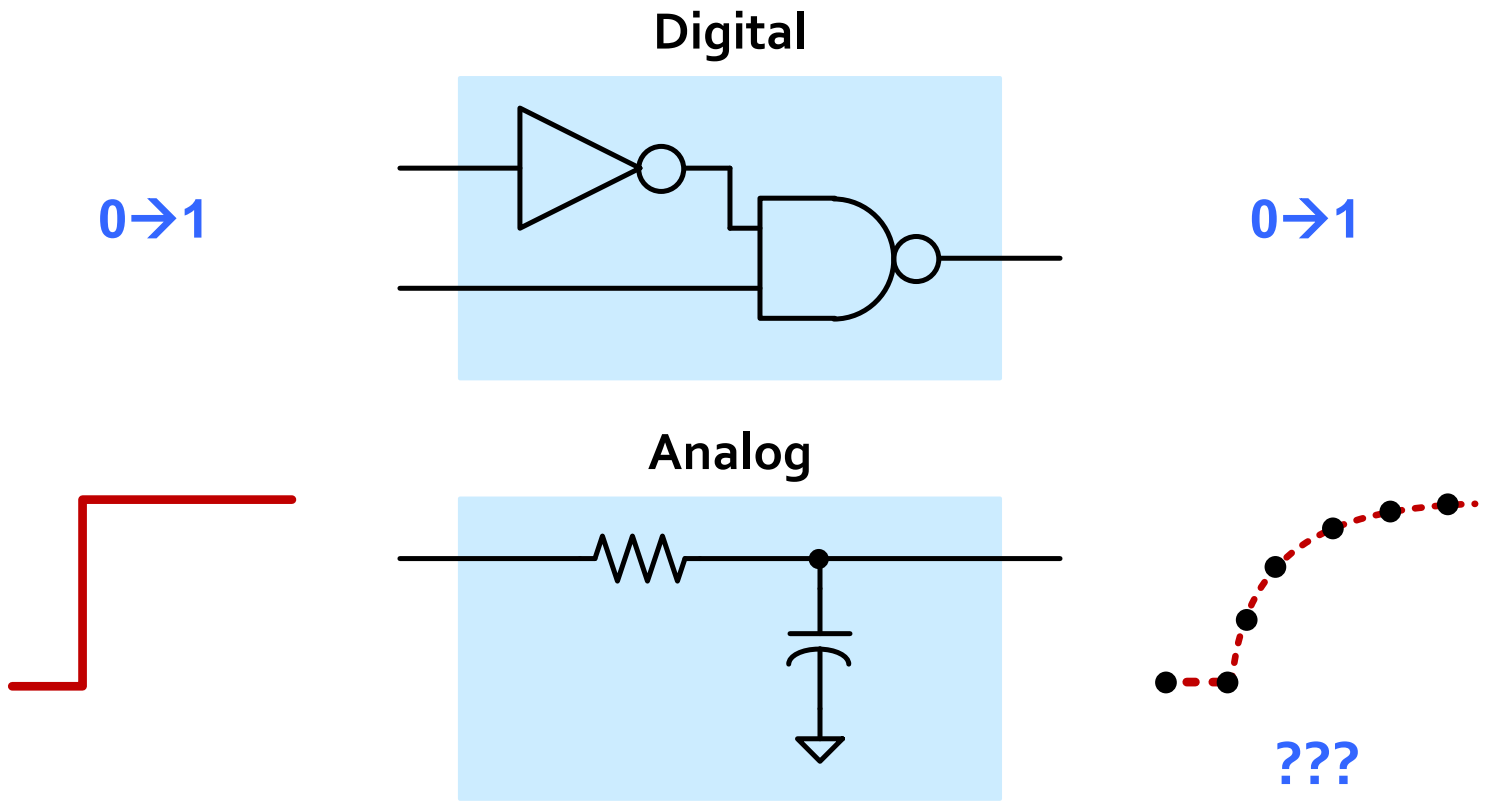
XMODEL Enables Analog in SystemVerilog

- *XMODEL* is a plug-in extension enabling ***fast and accurate analog/mixed-signal*** simulation in ***SystemVerilog***
  - ***Event-driven***: delivering 10~100x faster speed than Real-Number Model (RNM)
  - ***Analog***: supporting both functional and circuit-level models
  - ***SystemVerilog***: fully compliant with SystemVerilog-based flows (e.g. UVM)



Event-Driven Simulation of Analog

- How do we extend the Verilog's event-driven algorithm to simulating analog circuits?

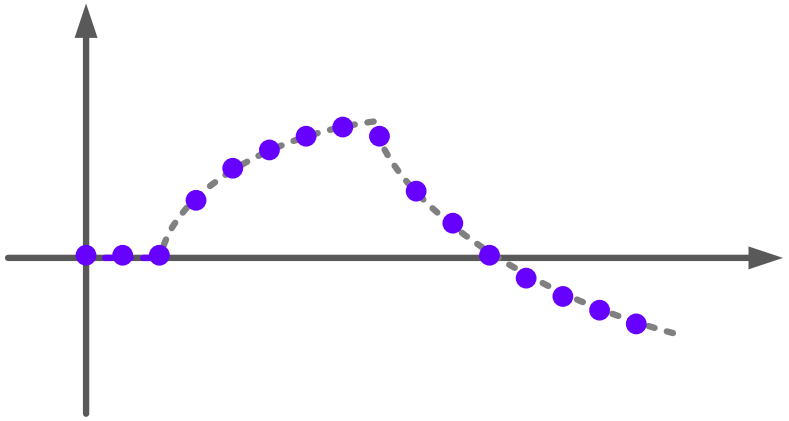


Expressing Analog Events

- *XMODEL* expresses analog signals in functional forms instead of using a series of time-value pairs:

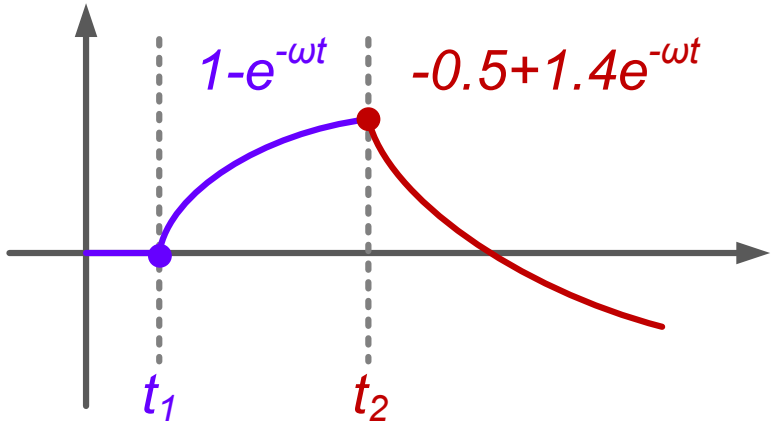
$$x(t) = \sum_i c_i t^{m_i} e^{-a_i t}$$

SPICE



Accuracy relies on fine time step

XMODEL



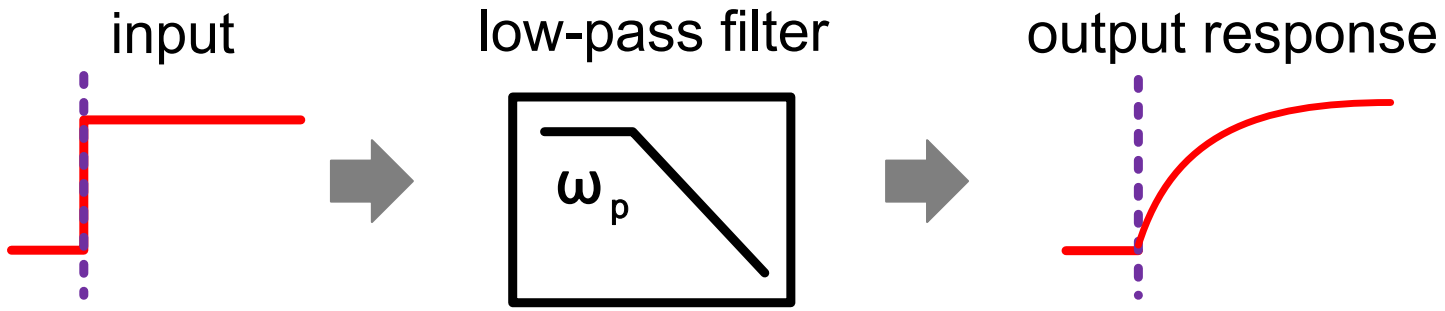
Events occur only when the coefficients are updated

Propagating Analog Events

- With the signals transformed into Laplace s-domain:

$$x(t) = \sum_i c_i t^{m_i - 1} e^{-a_i t} u(t) \xrightarrow{\mathcal{L}} X(s) = \sum_i \frac{b_i}{(s + a_i)^{m_i}}$$

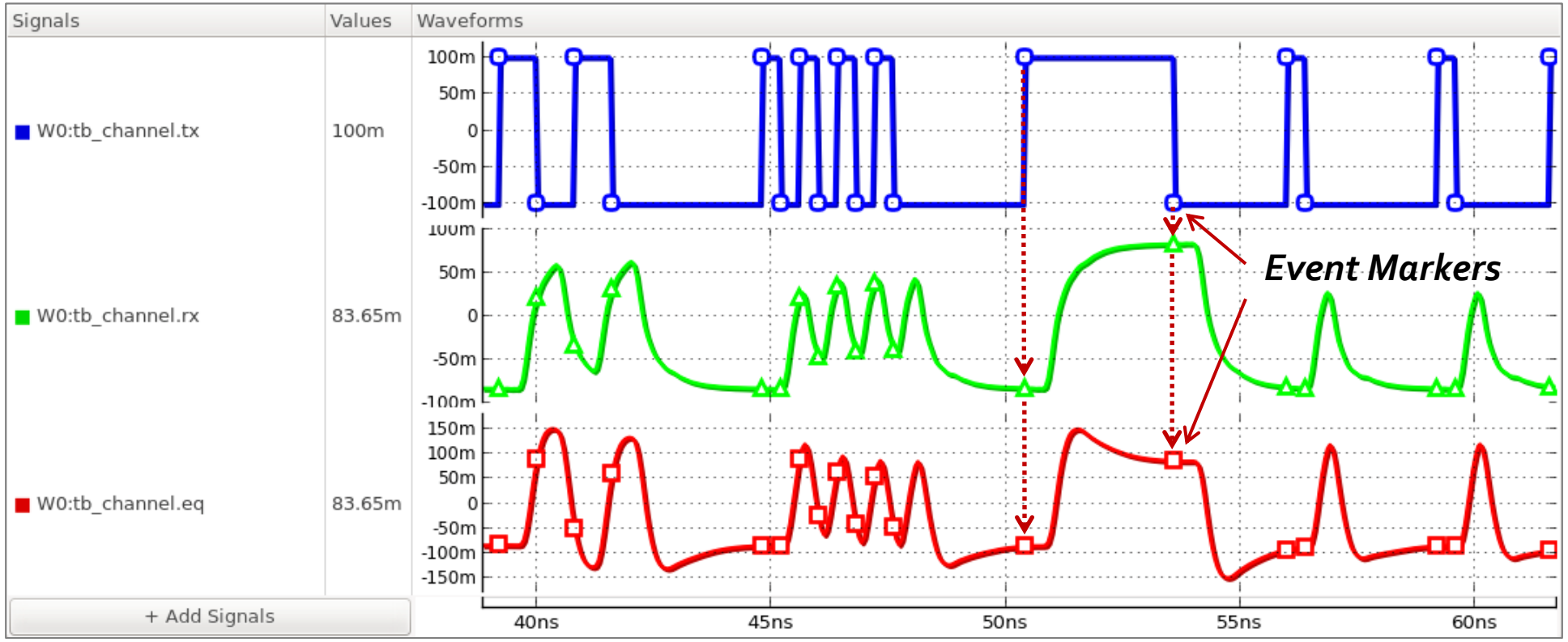
- The response of a system can be computed in an event-driven manner without time-step integration:



$$\frac{1}{s} \times \frac{\omega_p}{\omega_p + s} = \frac{1}{s} - \frac{1}{\omega_p + s}$$

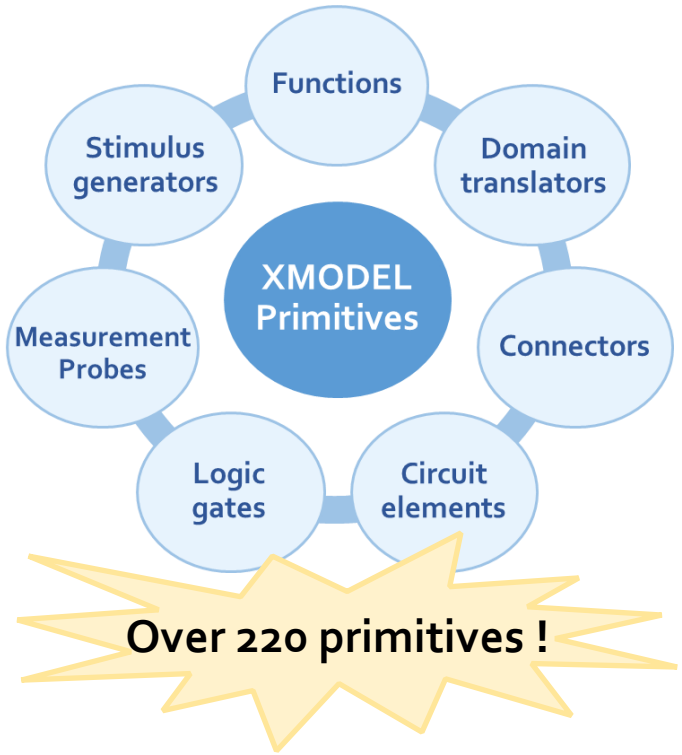
XMODEL's Event-Driven Simulation

- *XMODEL* is fast due to very few events triggered during the simulation



Composing Models with *XMODEL* Primitives

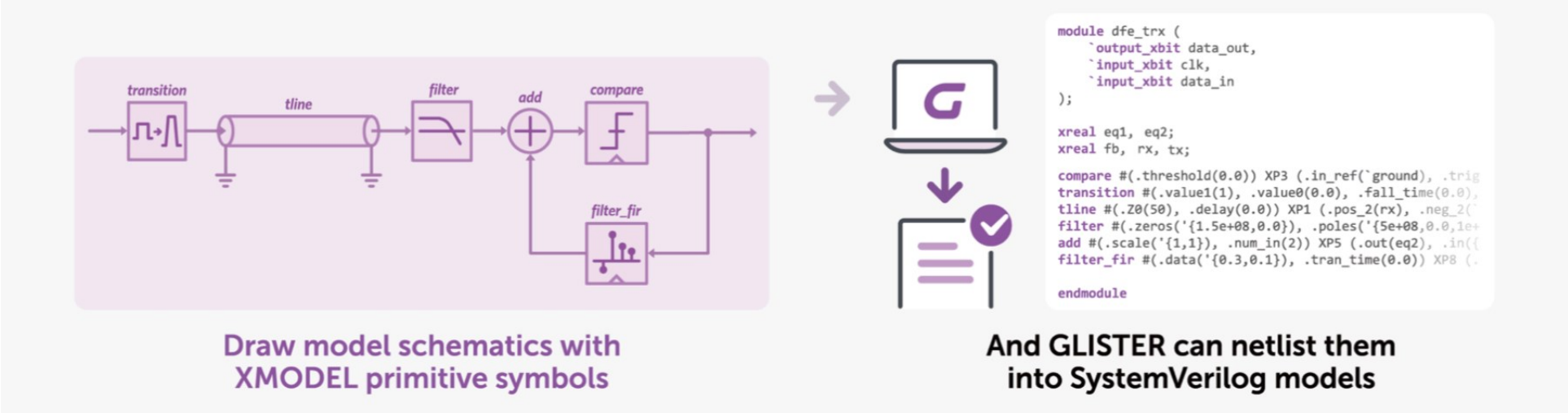
- With *XMODEL*, you don't have to write codes to model analog circuits
- Put together ***XMODEL primitives*** to describe their functions or circuits



Functions	Filters	Data conversion / comparison	Sampling, selection, and delay
add	filter	transition	sample
multiply	filter_var	slice	select
scale	filter_fir	compare	delay
pwl_func	filter_disc	dac	delay_var
poly_func	integ	adc	buffer
sin_func	integ_mod		
exp_func	integ_rst		
limit	deriv		
power			

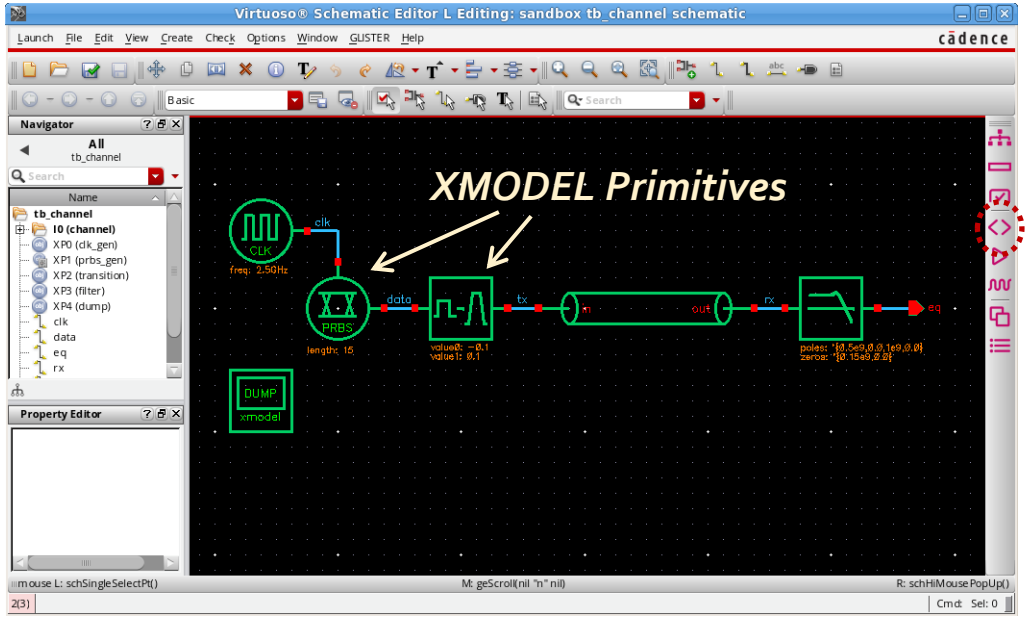
GLISTER: Build Top-Down Models

- ***GLISTER*** lets you build top-down analog models in Cadence Virtuoso or Synopsys CustomCompiler in schematic forms
- Simply place the *XMODEL* primitives on a schematic and connect them with wires; no coding is necessary!

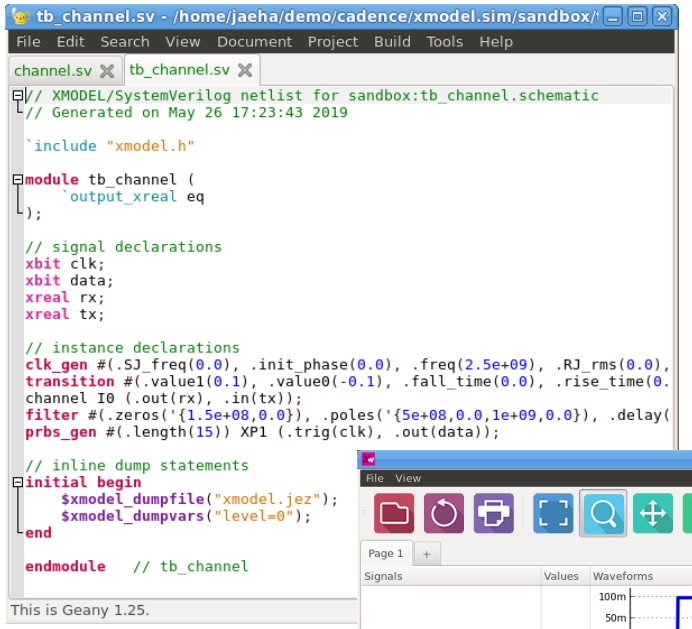


Top-Down Modeling with *XMODEL & GLISTER*

Describe top-down models in schematics using *GLISTER* & *XMODEL* primitives



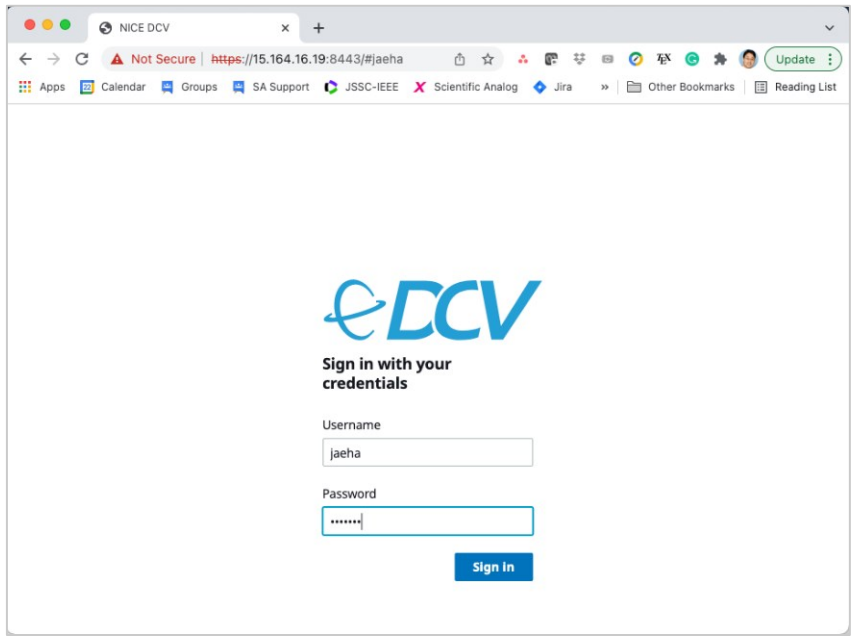
Netlist SystemVerilog models

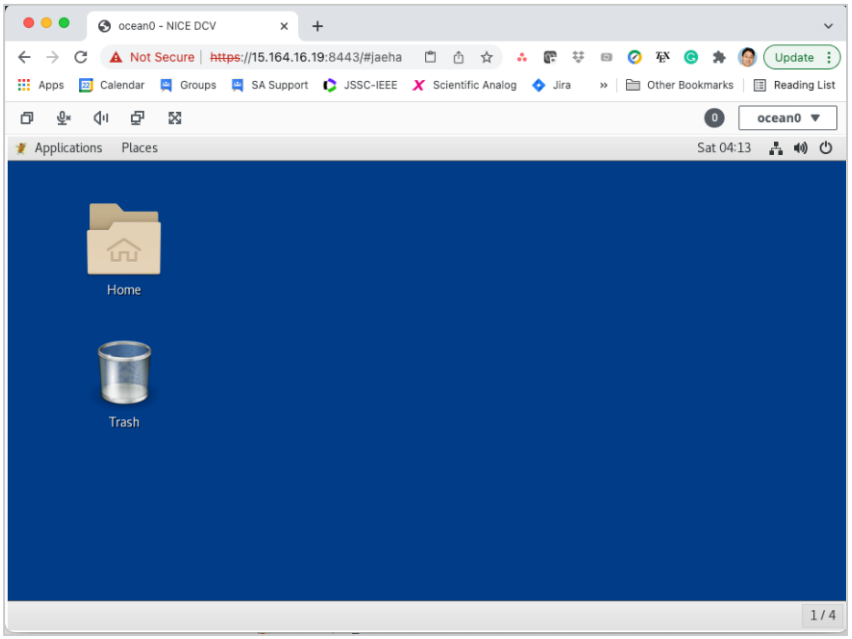


Simulate with *XMODEL*

Getting Ready

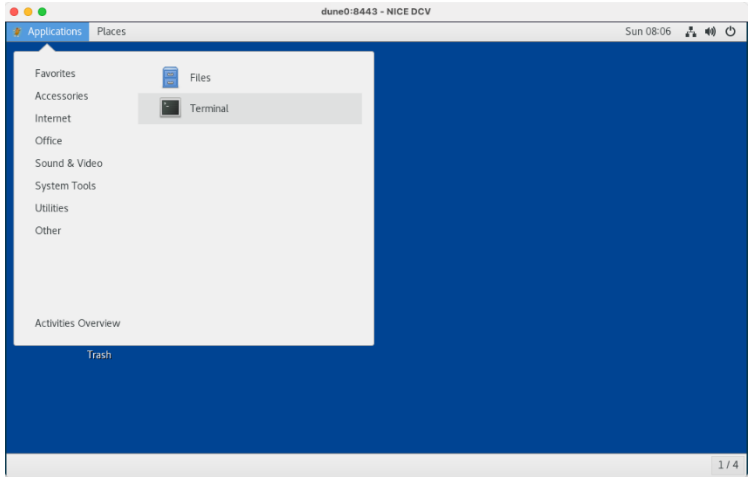
- Connect to the URL: **https://*<SERVER_IP>*:8443/#*<USERNAME>***
  - For example, https://111.22.33.44:8443/#usero1
  - You may have to ignore security warnings to proceed





Getting Ready (2)

- Open the "Terminal" App
  - Click the pull-down menu: *Applications→Favorites→ Terminal*

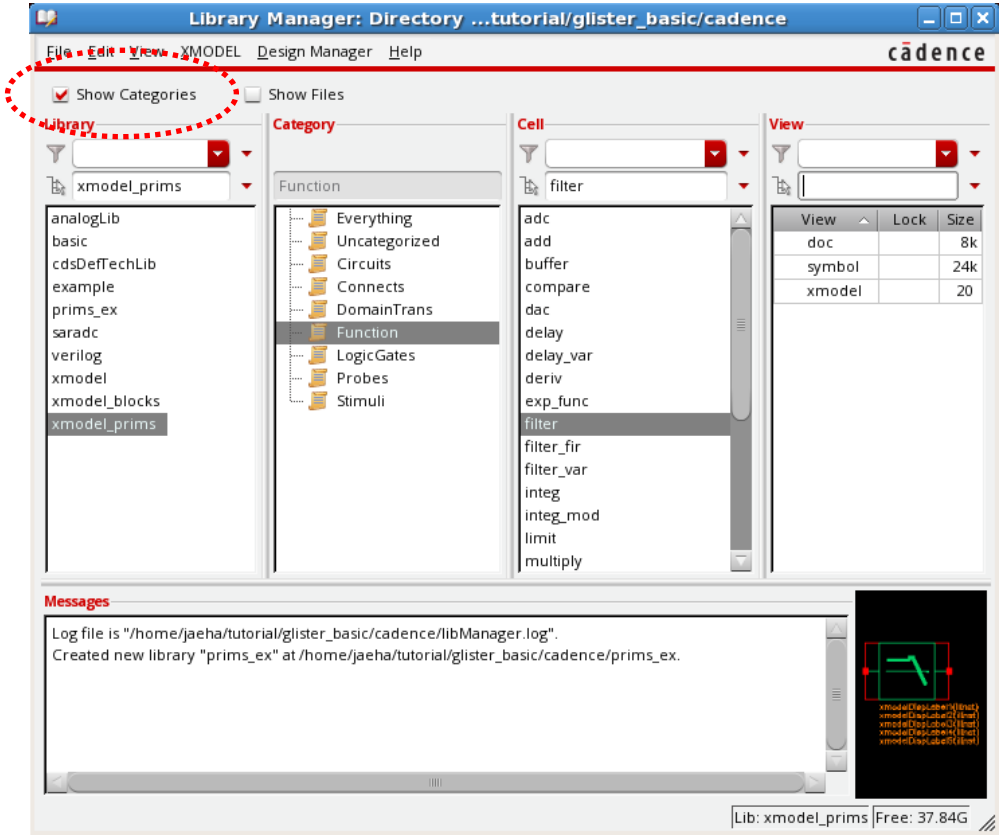


- And execute these commands to start Virtuoso:

```
$ tar zxvf UVM_XMODEL_202303.tar.gz
$ cd UVM_XMODEL
$ source etc/setup.bashrc
$ cd cadence
$ virtuoso &
```

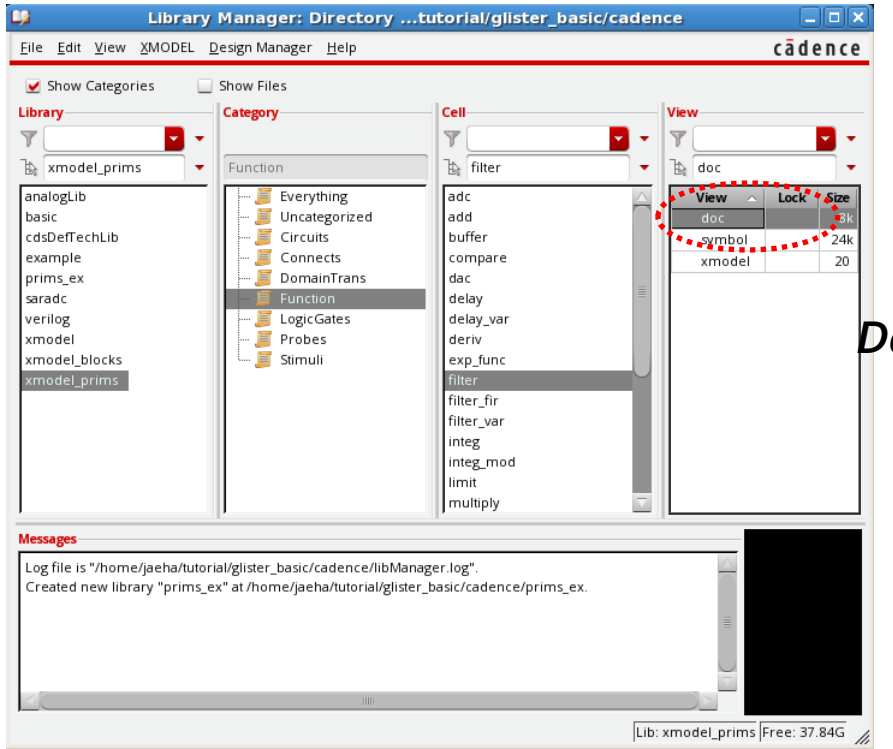
Browsing the *XMODEL* Primitive Library

- Click ***Tools*** → ***Library Manager*** in CIW
- Select ***xmodel_prims*** library and browse the *XMODEL* primitives available
- Check ***Show Categories*** option to browse primitives based on their categories

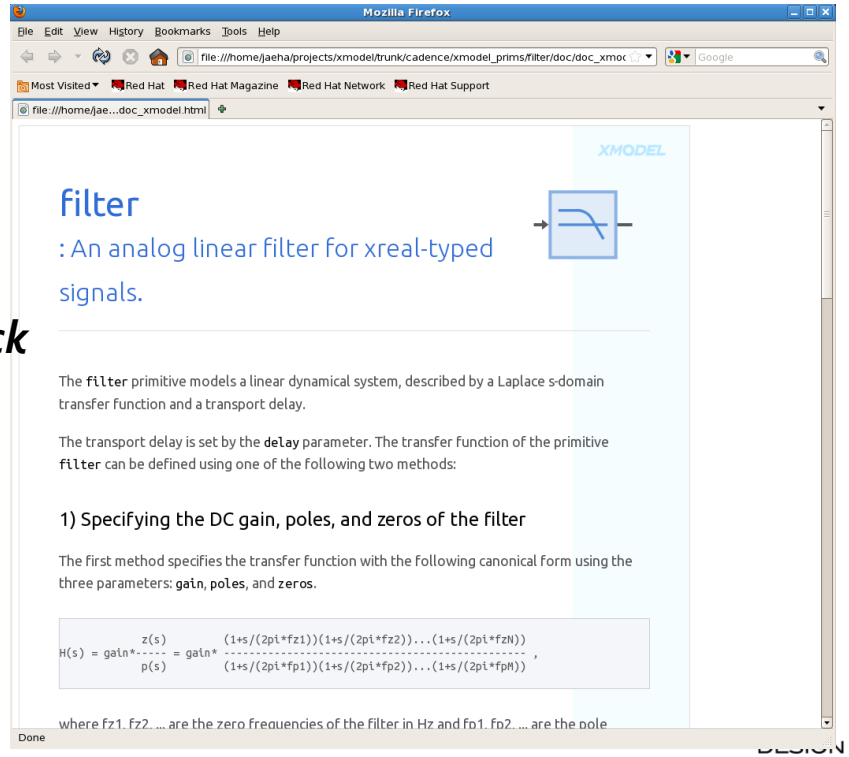


Accessing Documentations of *XMODEL* Primitives

- By double-clicking on the ***'doc'*** view of each primitive cell, or
- By typing ***'xmodel –h <primitive name>'*** on the command line



Double Click



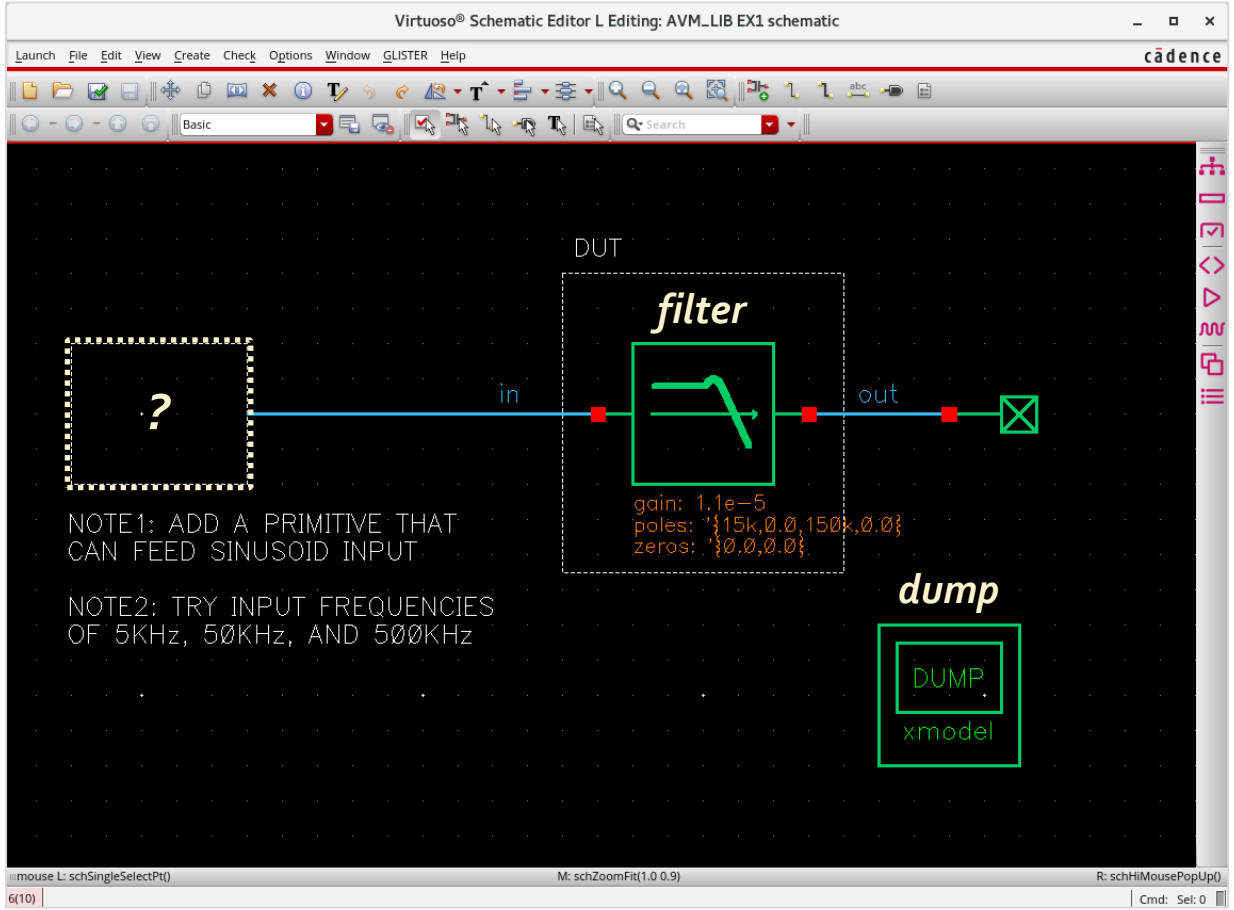
XMODEL Primitives At a Glance

Functions	**Describes functionality of analog/mixed-signal circuits** (add, multiply, deriv, integ, filter, select, power, pwl_func, poly_func, transition, sample, compare, dac, adc, …)
Circuits	**Represents circuit elements** (resistor, capacitor, inductor, switch, diode, nmosfet, pmosfet, vsource, isource, vcvs, vccs, ccvs, cccs, …)
Logic Gates	**Models digital logic gates with xbit input/outputs** (buf_xbit, inv_xbit, nand_xbit, nor_xbit, xor_xbit, mux_xbit, dff_xbit, …)
Domain Translators	**Converts between a clock and one of its properties such as frequency, phase, period, duty-cycle, and delay** (clk_to_freq, clk_to_phase, freq_to_clk, phase_to_clk, …)
Connect	**Make connections between signals with different types** (xbit_to_bit, bit_to_xbit, xreal_to_real, real_to_xreal, …)
Stimuli	**Generate stimulus signals** (dc_gen, sin_gen, pwl_gen, clk_gen, prbs_gen, …)
Probes / Measures	**Make measurements on the simulated waveforms** (probe_freq, probe_delay, dump, meas_avg, meas_pp, trig_cross, trig_rise, …)

Lab Exercise #1

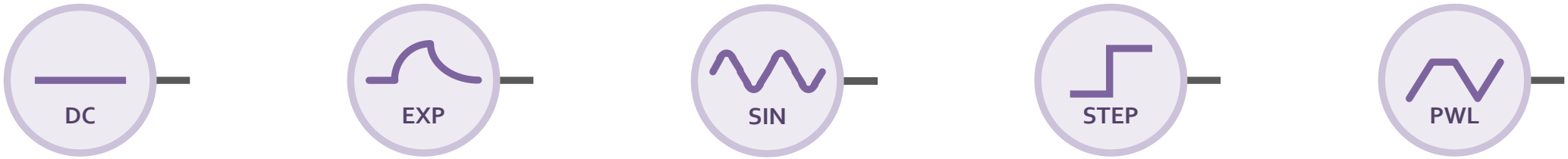
- Open ***AVM_LIB.EX1:schematic*** cellview

- Place a proper primitive symbol in the empty box to supply a ***sinusoid input*** to the filter DUT



Hint for Lab Exercise #1

- Choose your answer from one of these ***stimulus generator primitives***:

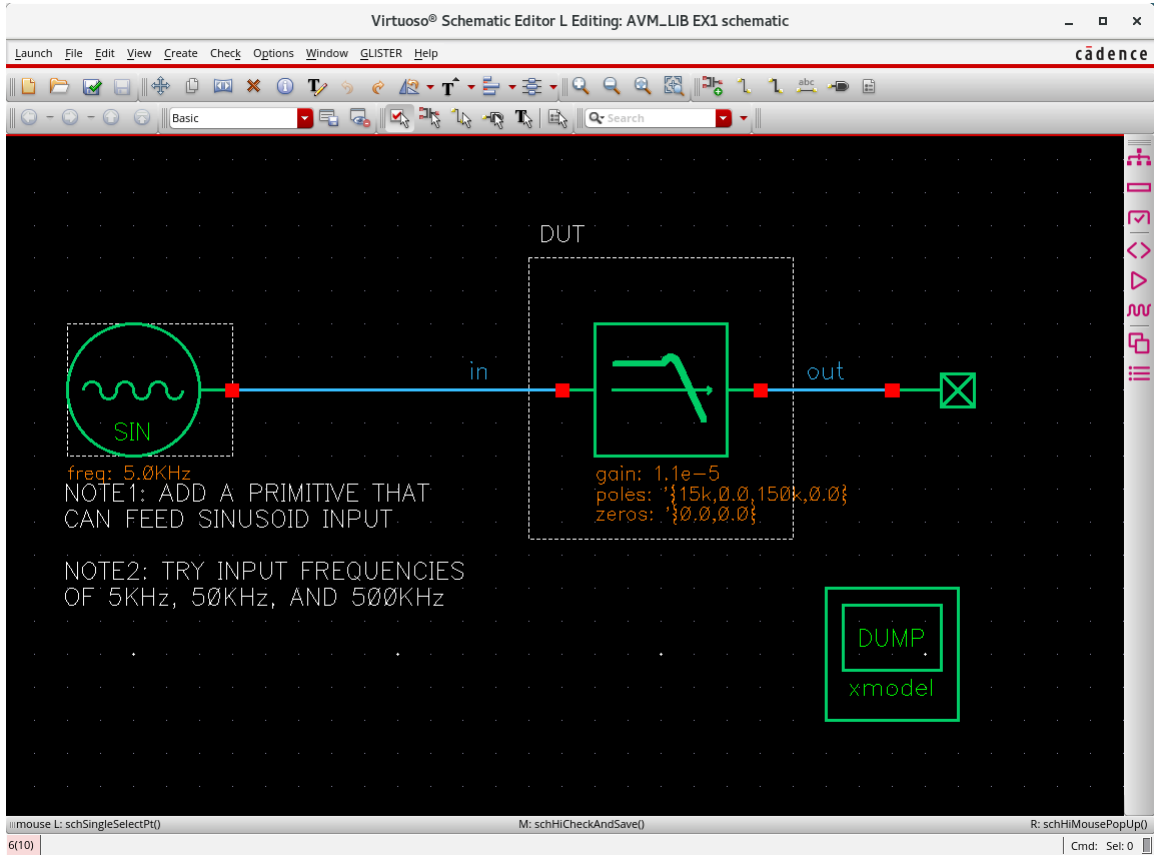


dc_gen ***exp_gen*** ***sin_gen*** ***step_gen*** ***pwl_gen***

Solution for Lab Exercise #1

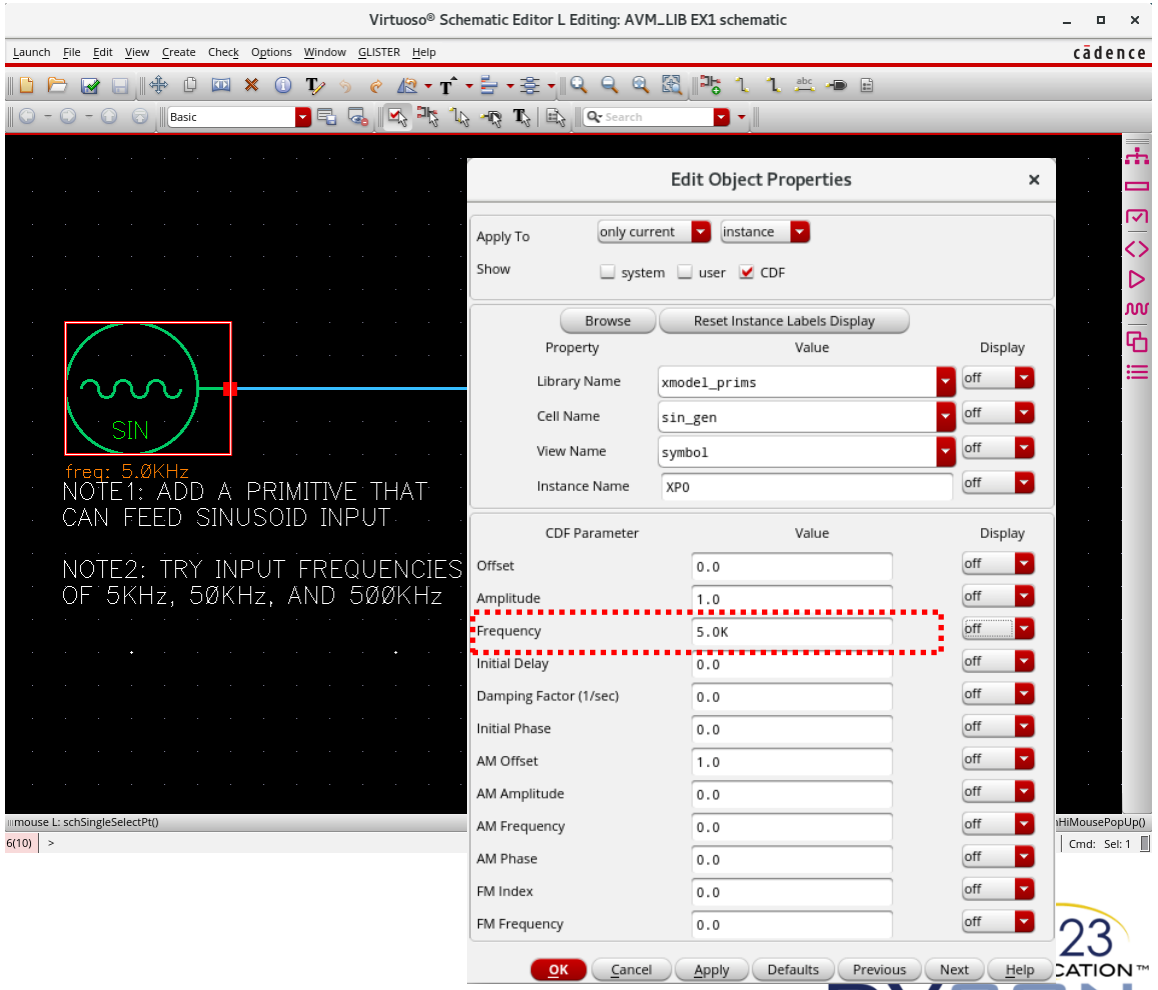
- The answer is "***sin_gen***"

- To place an instance of ***sin_gen***:
  - Select ***Create→Instance*** (or press 'I')
  - Choose ***xmodel_prims:sin_gen.symbol***
  - Place the symbol on the schematic



Editing Instance Parameters

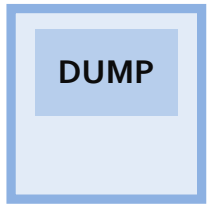
- To edit its parameters (e.g. *frequency*):
  - Select the symbol on the schematic
  - Click right mouse button and select ***Properties…*** from the pop-up menu (or press '*Q*')
  - Edit the parameter values
  - Click '*OK*'



More *XMODEL* Primitives Explained

filter

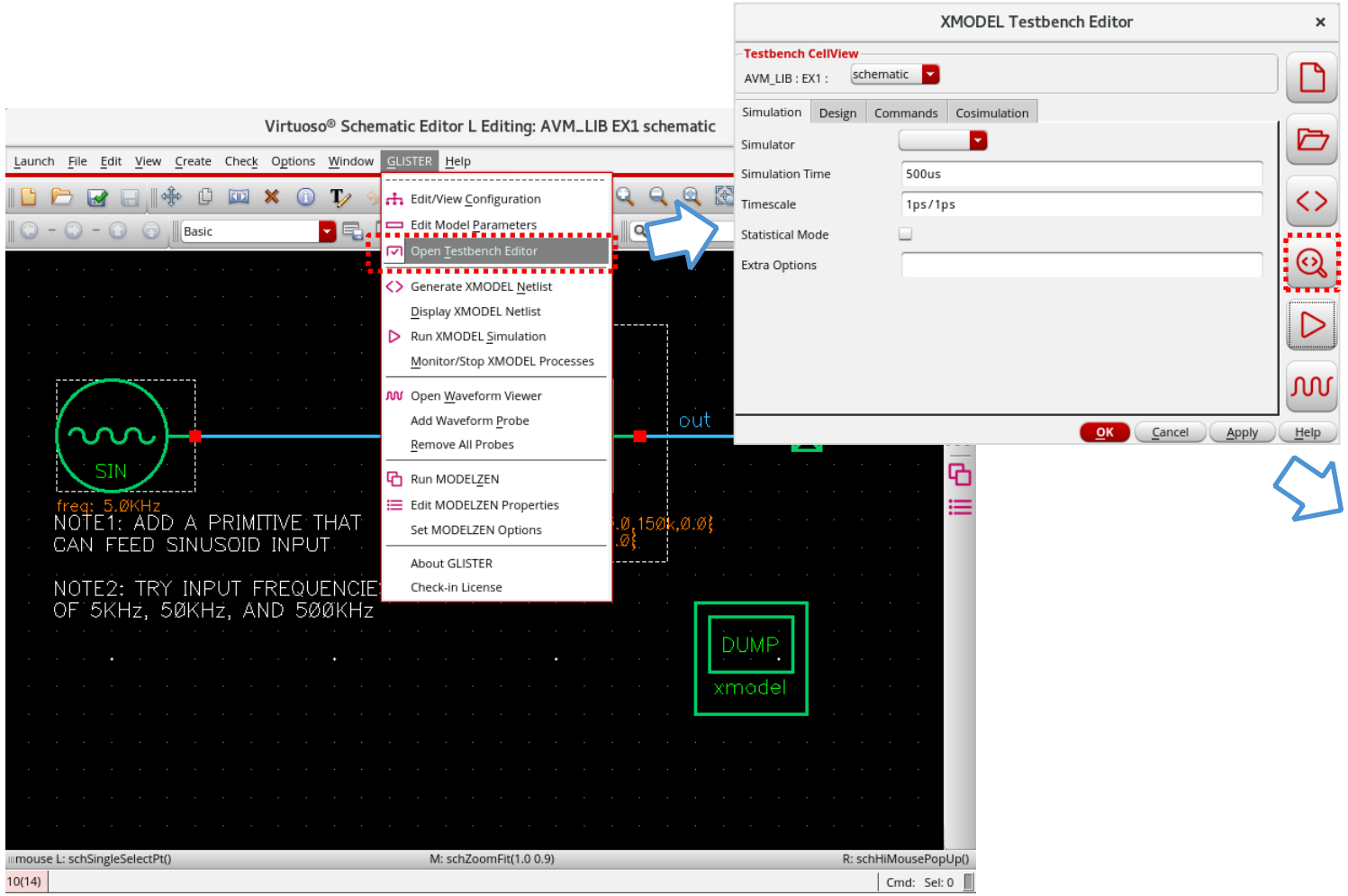
- Models a linear analog filter with its gain, poles, and zeros
- This example models a bandpass filter with two poles at 15kHz & 150kHz and a zero at DC (`poles='{15k,0.0,150k,0.0}, zeros='{0.0,0.0}`)



dump

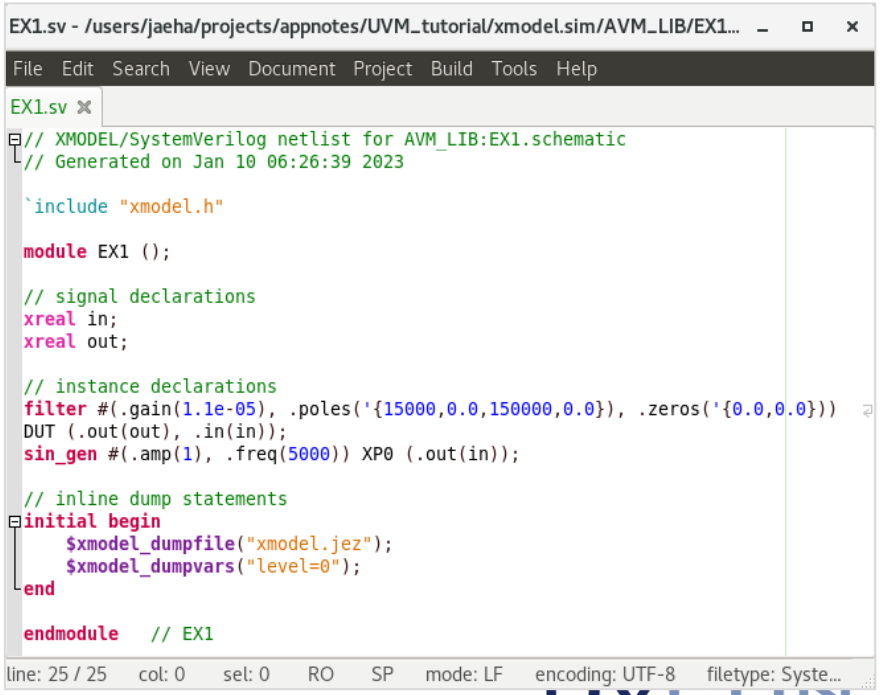
- Instructs the simulator to record the simulated waveforms
- Supported file format: JEZ and FSDB
- Various options available (e.g. level of monitoring depth)

Generating *XMODEL* Netlist



← *Generate Netlist*

← *Display Netlist*



XMODEL Signal Types

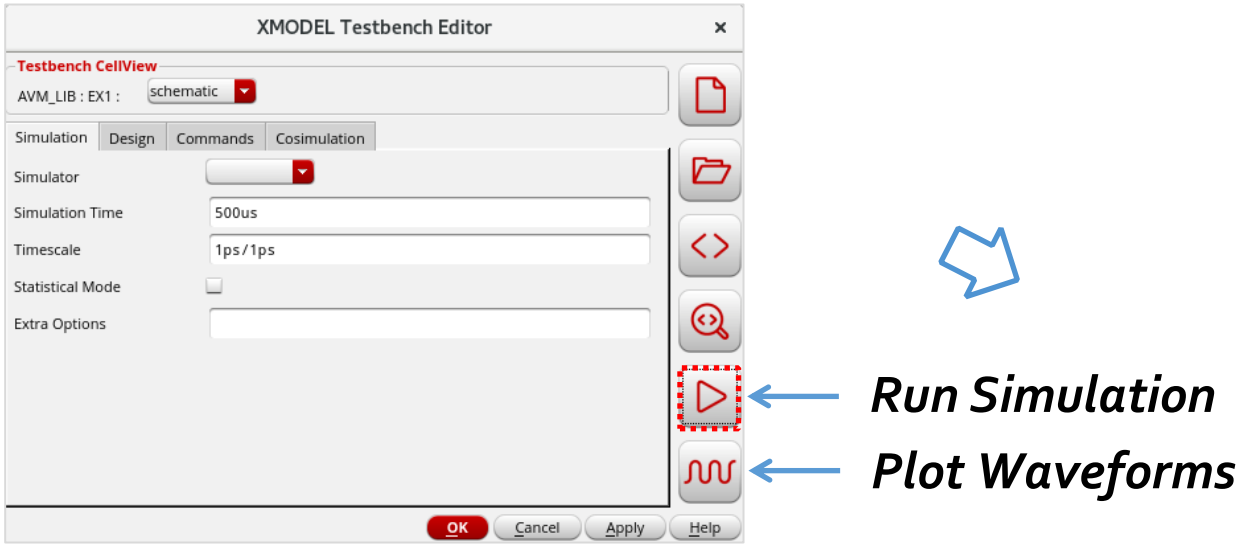
- *XMODEL* introduces two new data types:

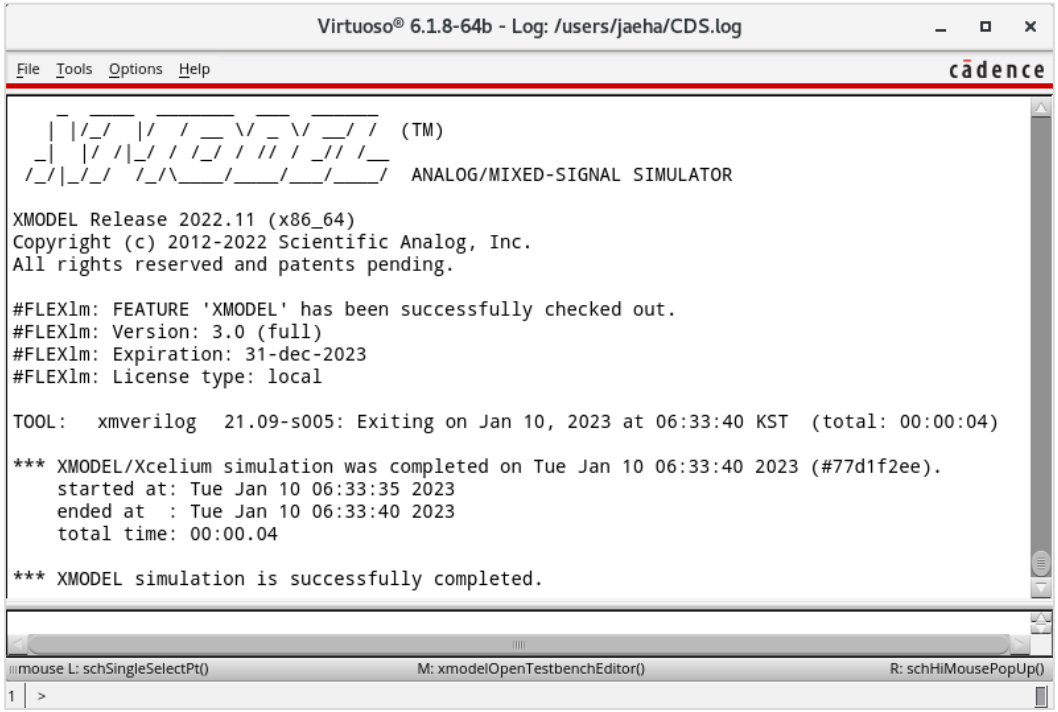
  xreal : for continuous-time analog signals in event-driven format

  xbit : for *timing-accurate* digital signals (e.g. clocks or pulses)

- *GLISTER* can automatically detect the type of each signal during netlisting and insert type-coercing connectors as necessary

Running *XMODEL* Simulation





- Note: this process is equivalent to executing:

```
$ cd $XMODEL_SIMDIR/AVM_LIB/EX1/schematic
$ make runsim
```

The *xmodel* Launcher Script

- *XMODEL* simulations are basically SystemVerilog with additional libraries and we use a wrapper script called ***'xmodel'*** to provide consistent interface with different SV simulators (Xcelium, VCS, Questa)
- Basic usage:

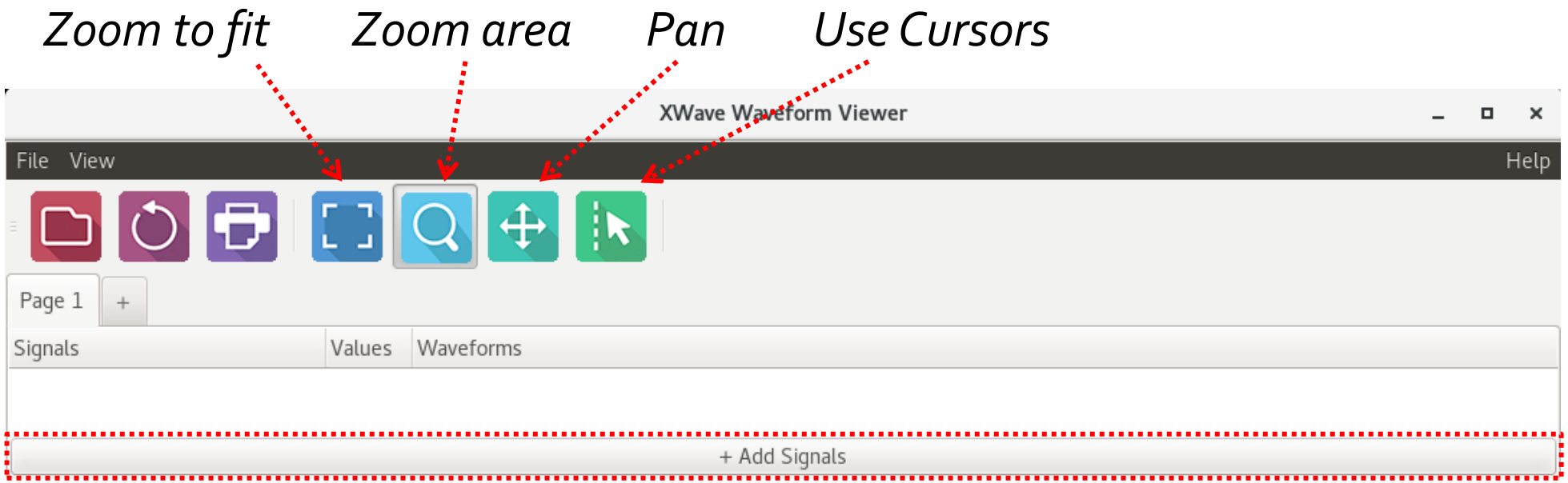
```
$ xmodel EX1.sv --top EX1 --simtime 500us --simulator [vcs|xcelium|questa]
```

- Use '--command' option to see the actual commands executed, e.g.:

```
$ xmodel EX1.sv --top EX1 --simtime 500us --simulator vcs --command
```

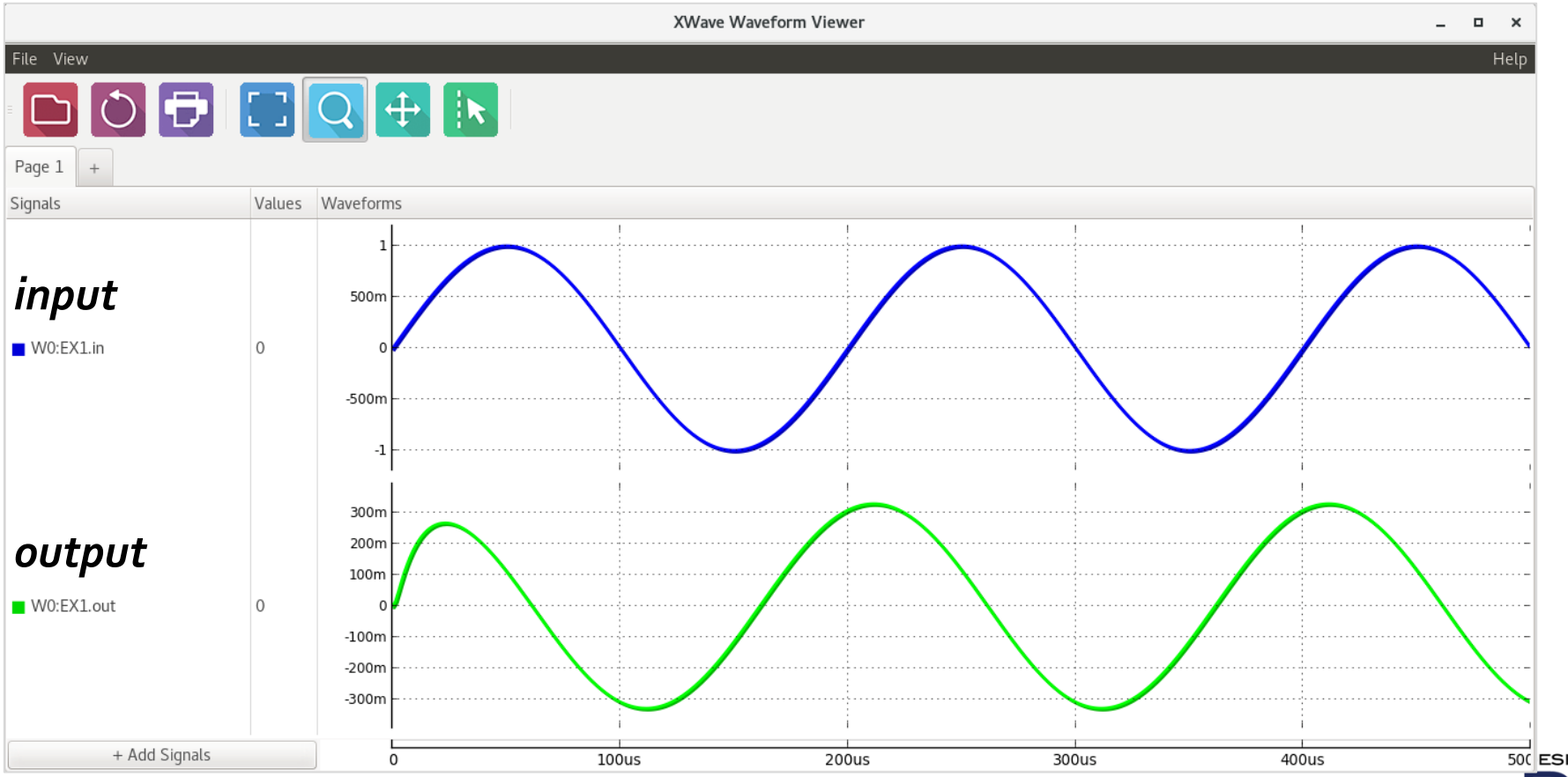
Viewing Waveforms

- Click  *Open Waveform Viewer to* start *XWAVE*
- Click "+ *Add Signals*" icon on the bottom to browse & select signals



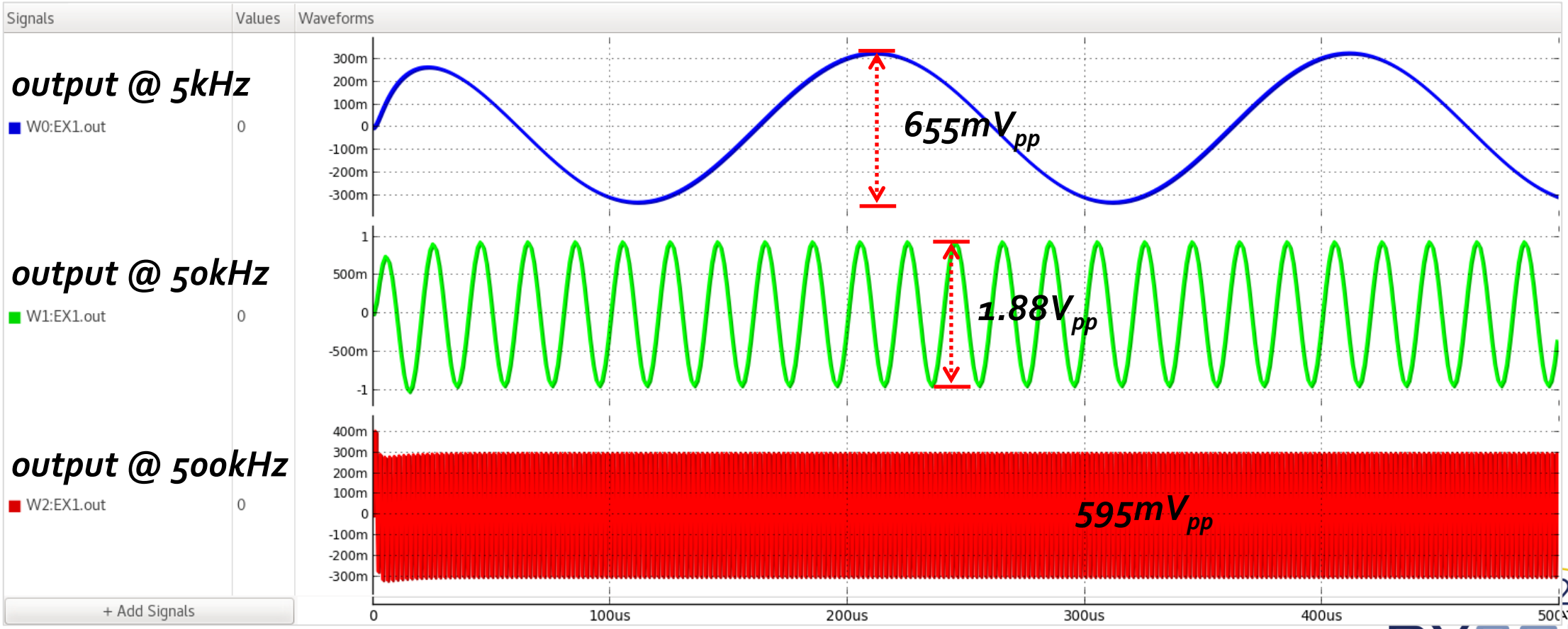
Simulated Waveforms

- For a 5kHz input with $2V_{pp}$ swing (-1~+1V), the output has ~$655mV_{pp}$ swing



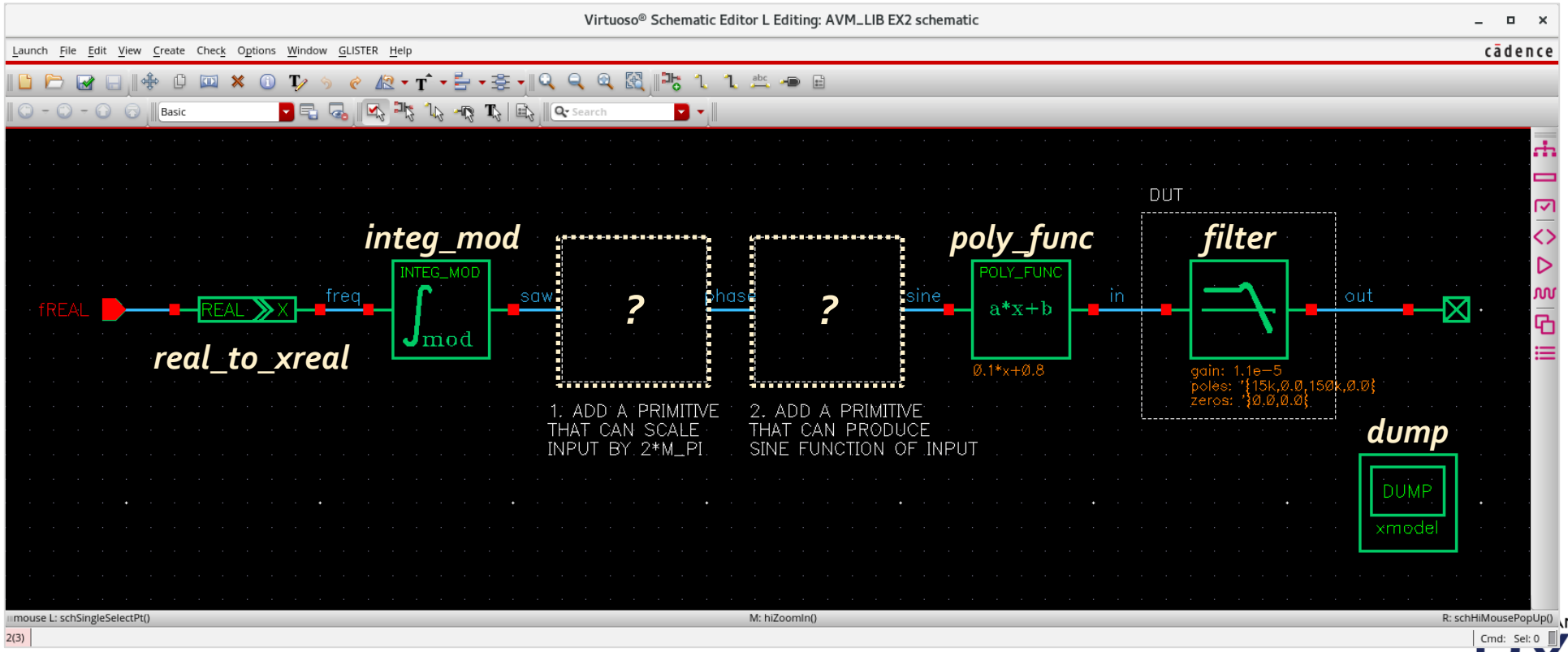
Lab Exercise #1 (cont'd)

- Try different input frequencies and check how the output swing varies

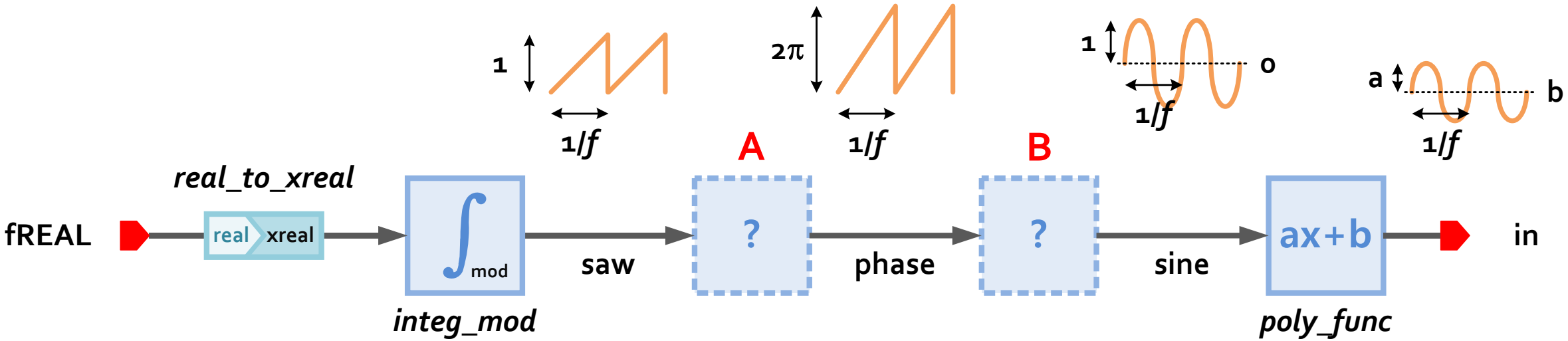


Lab Exercise #2

- We want a real-type input ***fREAL*** to control the input sinusoid frequency
- Place proper primitives in the empty boxes of ***AVM_LIB.EX2:schematic***



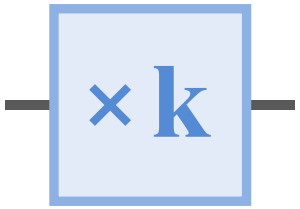
Lab Exercise #2 Explained



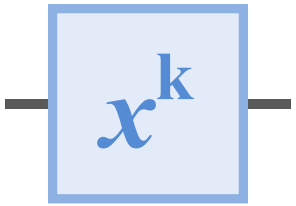
- ***real_to_xreal*** primitive converts a real-type input ***fREAL*** to an xreal-type signal
- ***integ_mod*** primitive produces a sawtooth signal **saw** with a frequency equal to **fREAL** and an amplitude equal to 1 by computing ***integral*(*x*) *modulo* 1**
- We want primitive **A** to scale **saw** by 2π to produce a phase sweeping signal ***phase***
- We want primitive **B** to produce a sinusoidal wave from ***phase***

Hint for Lab Exercise #2

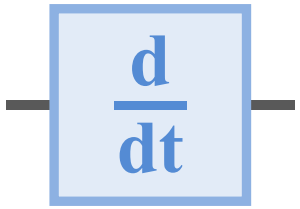
- Choose your answers from these ***function primitives***:



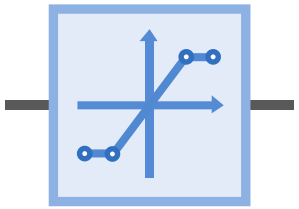
scale



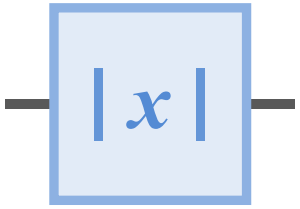
power



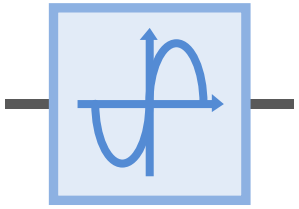
deriv



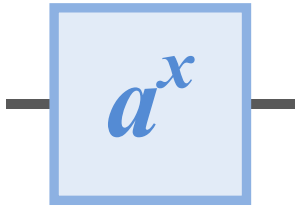
pwl_func



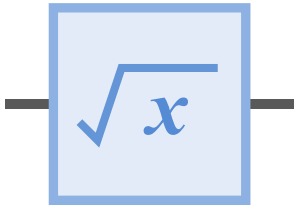
abs_func



sin_func



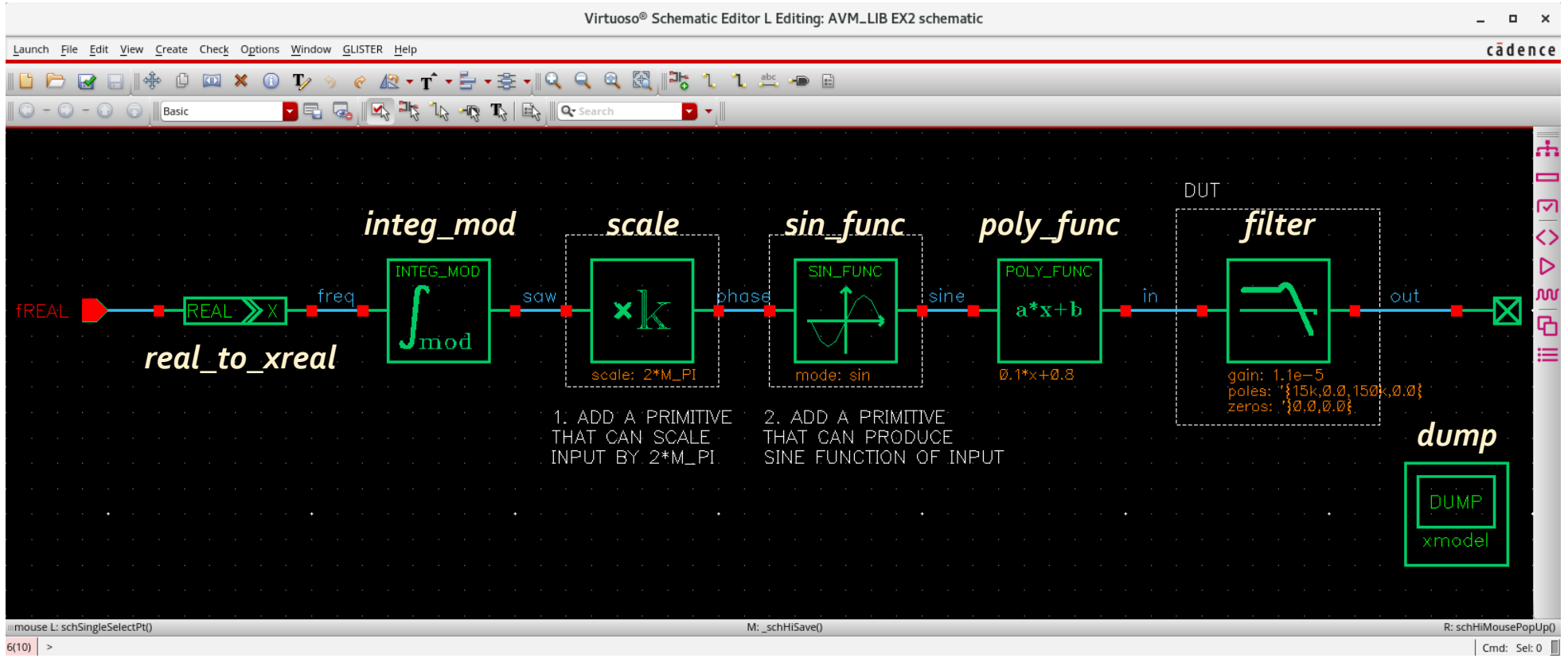
exp_func



sqrt_func

Solution for Lab Exercise #2

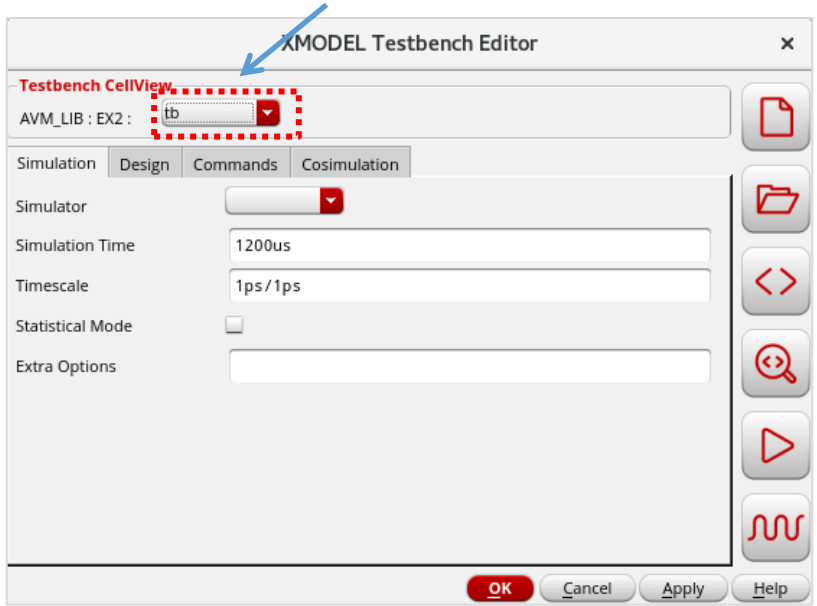
- The answers are: "***scale***" (with scale factor of `2*M_PI`) and "***sin_func***"

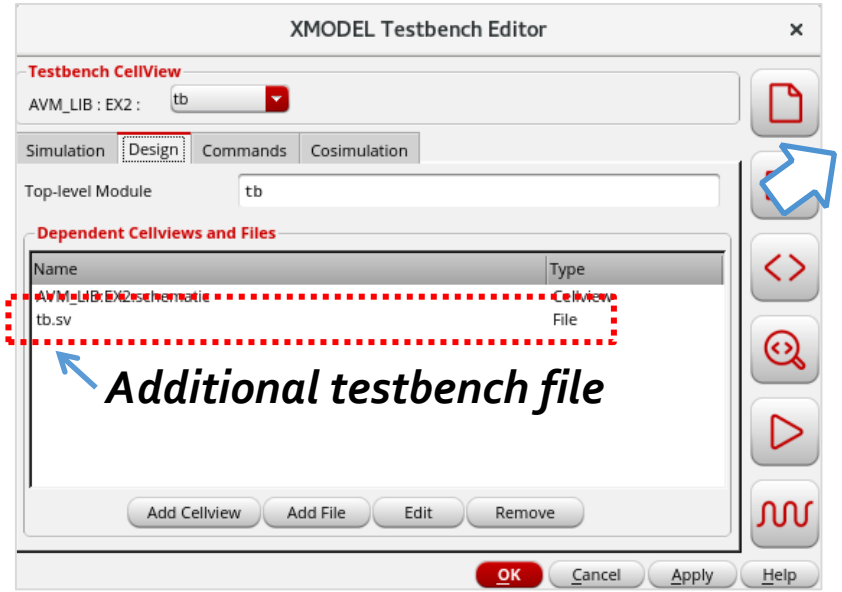


XMODEL Testbench View

- Can define testbench settings with customized set of files and commands
- Testbench view ***AVM_LIB.EX2:tb*** adds a top-level testbench source file that gradually increases ***fREAL*** from 5kHz to 500kHz

Choose TB view named 'tb'



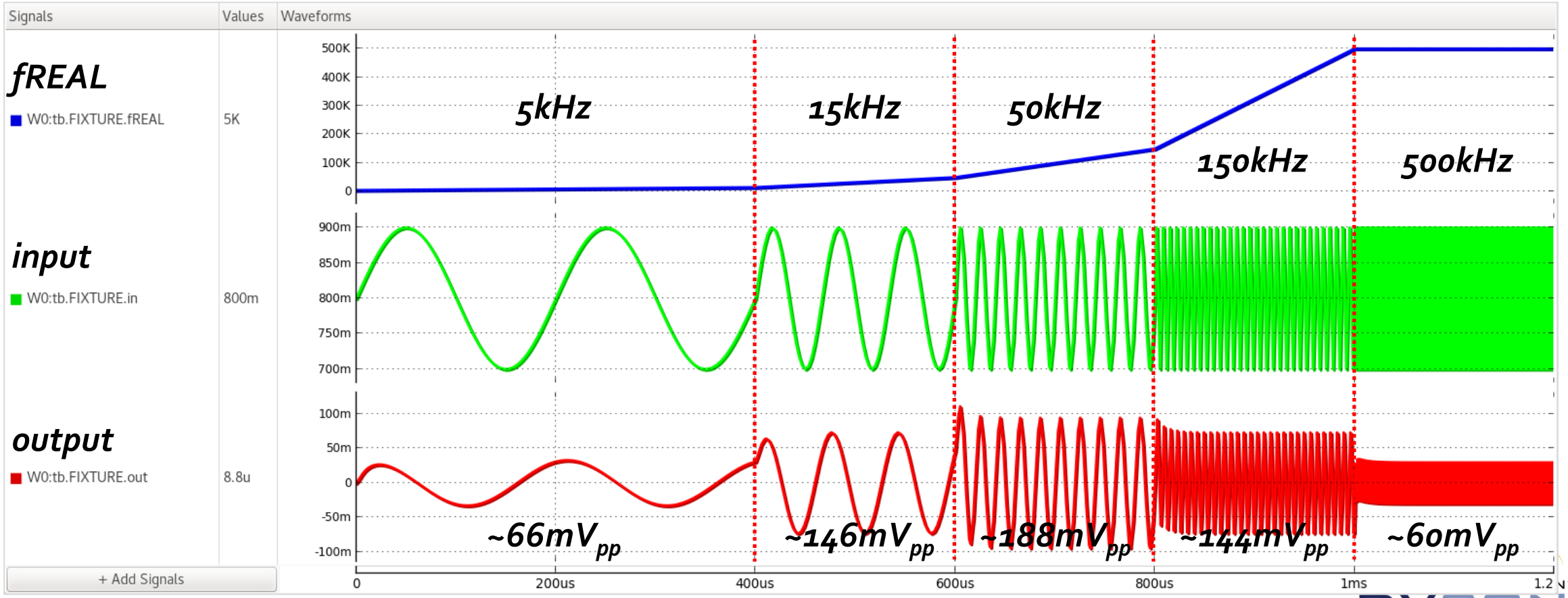


tb.sv

```
module tb ();

real fREAL;

// DUT FIXTURE
EX2     FIXTURE (.fREAL(fREAL));

// input stimulus
initial begin
    fREAL = 5.0e3;              // 5kHz
    #(400us);
    fREAL = 15.0e3;             // 15kHz
    #(200us);
    fREAL = 50.0e3;             // 50kHz
    #(200us);
    fREAL = 150.0e3;            // 150kHz
    #(200us);
    fREAL = 500.0e3;            // 500kHz
    #(200us);
end

endmodule
```

Simulated Waveforms

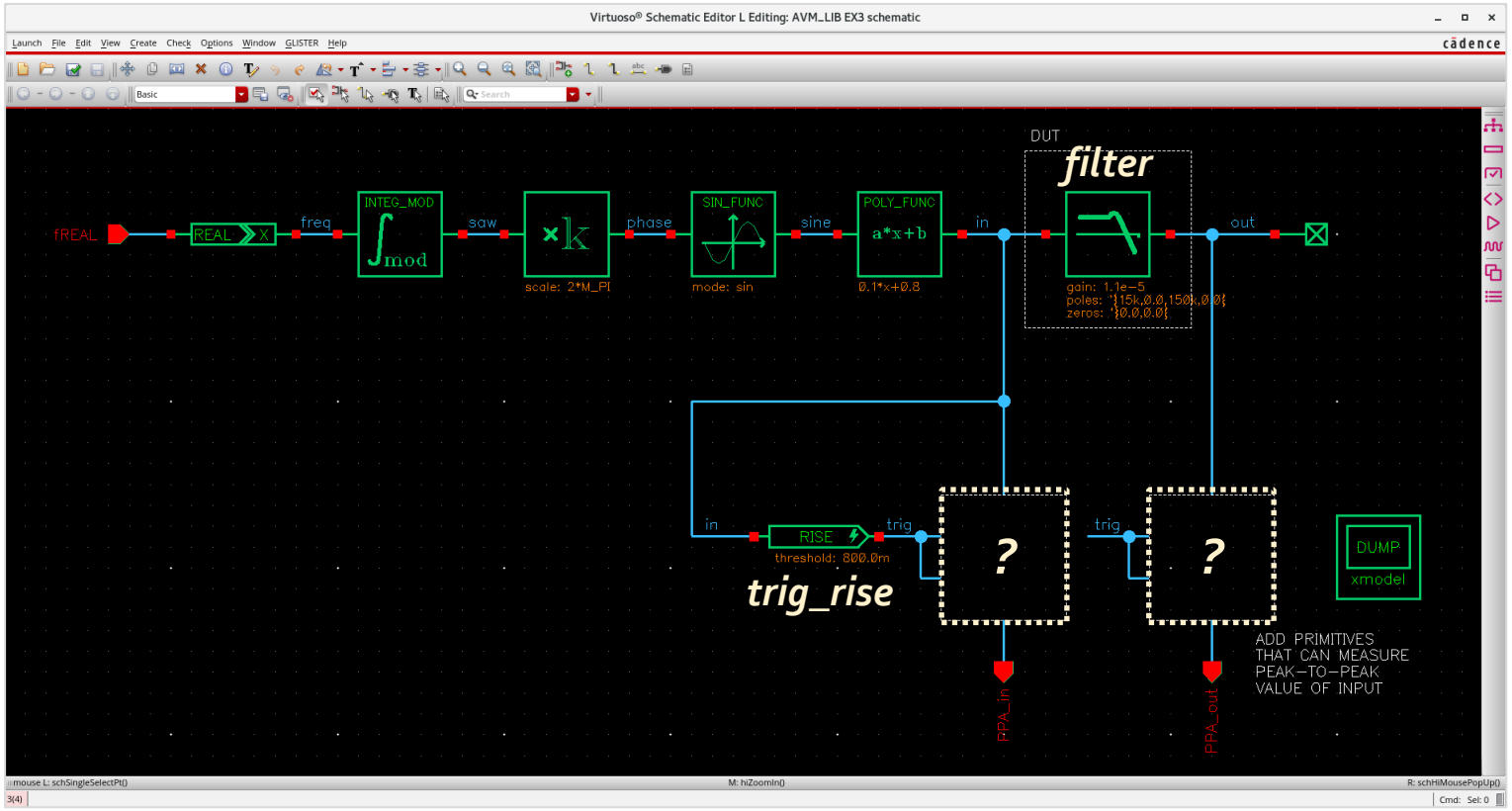
- The filter output shows different amplitude for each input frequency value



Lab Exercise #3

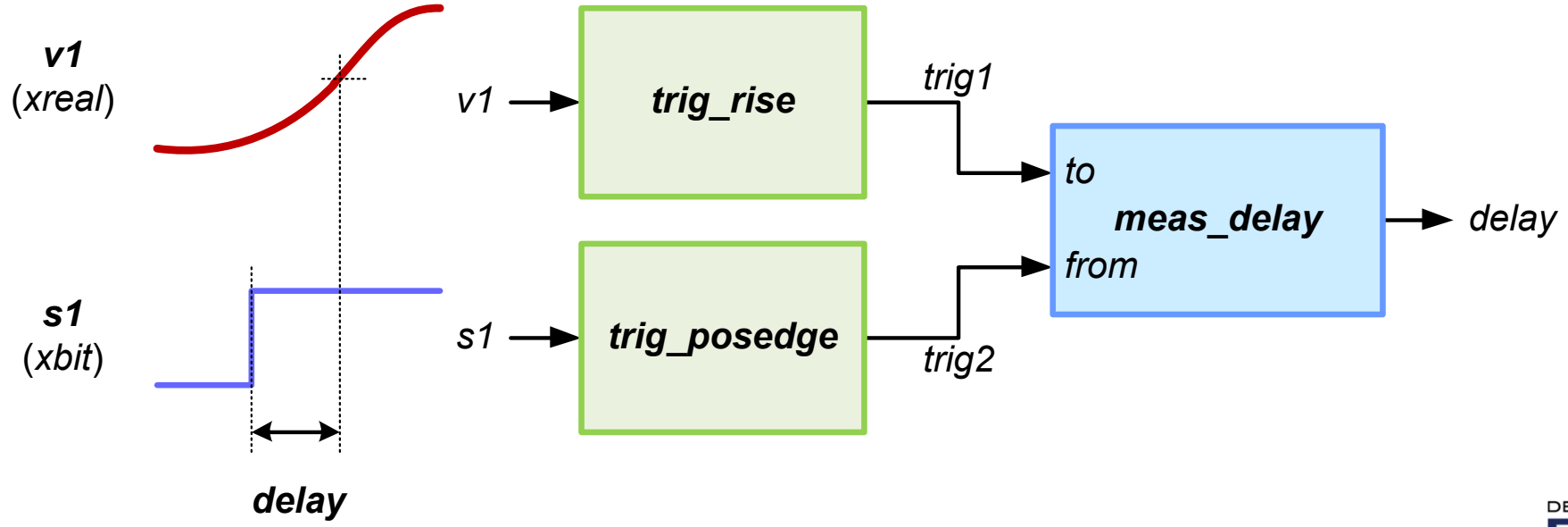
- Open ***AVM_LIB.EX3: schematic*** cellview

- Place proper primitives in the empty boxes to ***measure the peak-to-peak swings*** of the filter's input & output



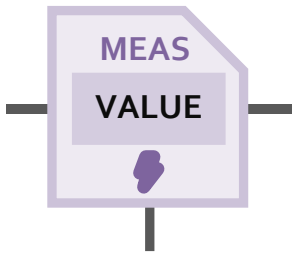
Measurement Primitives of *XMODEL*

- Measurement primitives measure the characteristics of signals at a time instant or over a time interval indicated by trigger signals
  - A variety of measurements are possible by combining ***trigger & measure*** primitives
  - e.g. measuring the delay from s1's positive edge to v1's rising

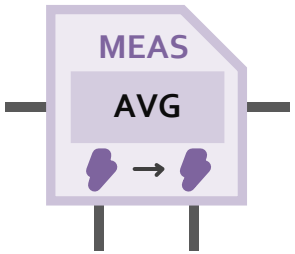


Hint for Lab Exercise #3

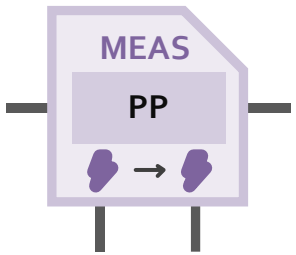
- Choose your answer from one of these ***measurement primitives***:



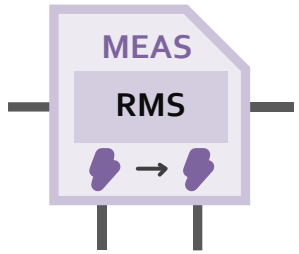
meas_value



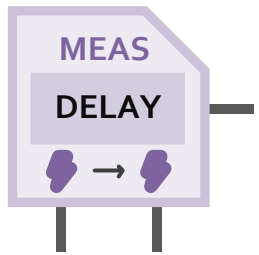
meas_avg



meas_pp



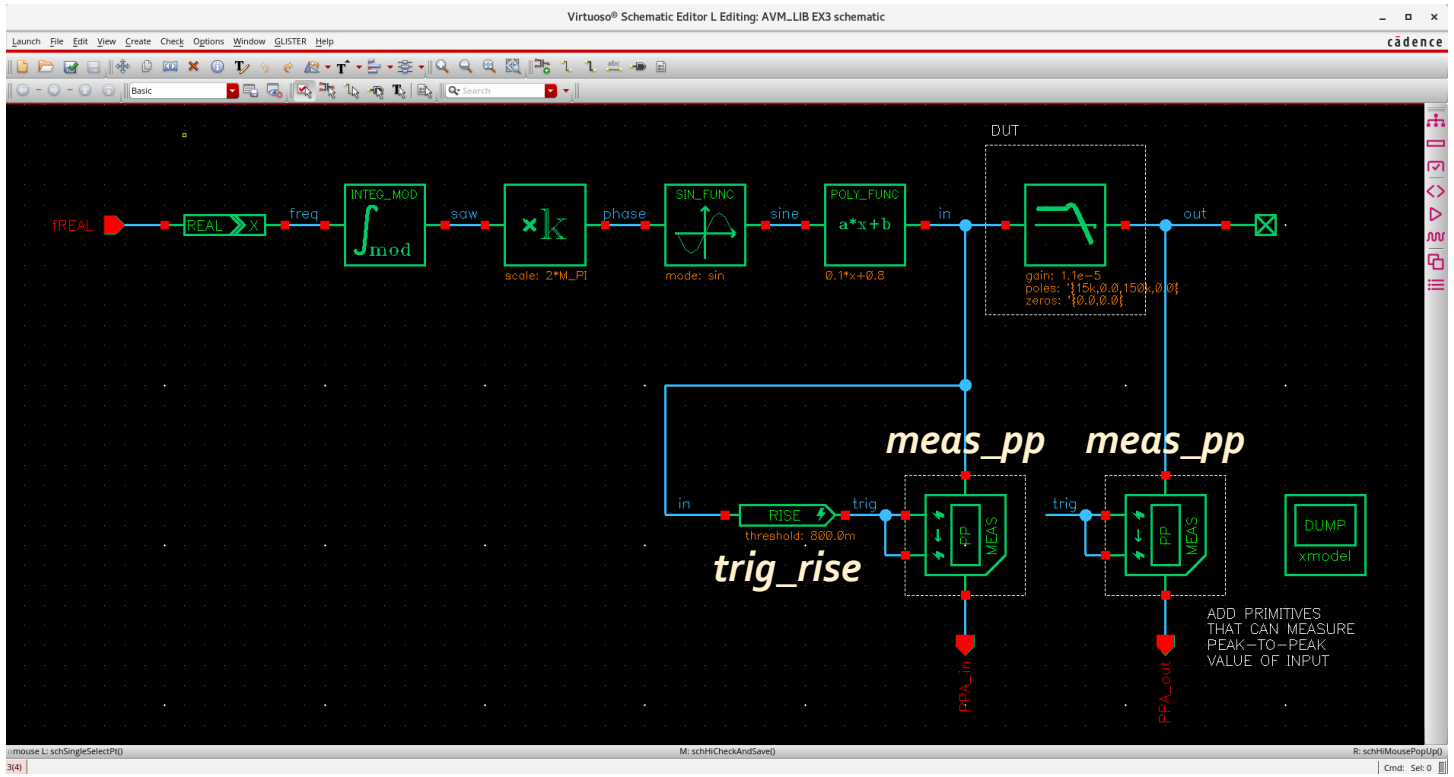
meas_rms



meas_delay

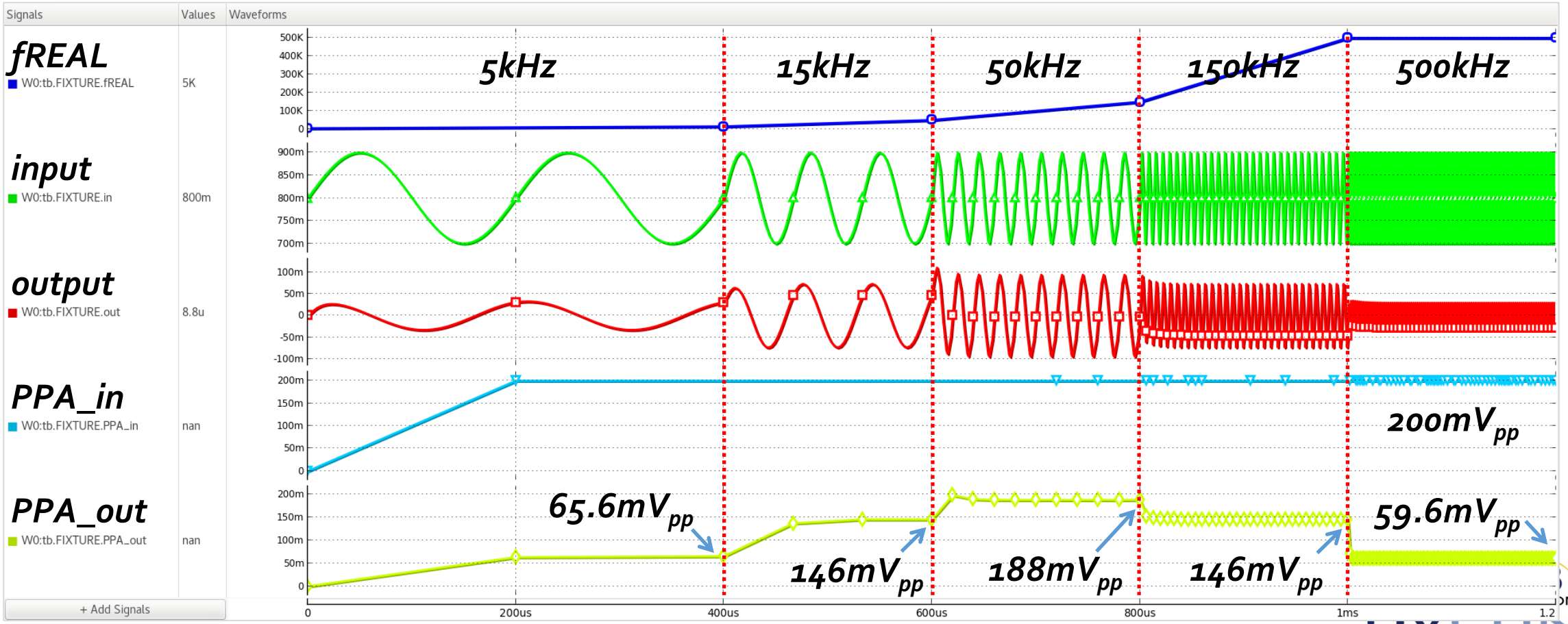
Solution for Lab Exercise #3

- The answer is "***meas_pp***"
- ***trig_rise*** primitive produces a signal ***trig*** triggering every period
- ***meas_pp*** primitives with ***from/to*** timings triggered by ***trig*** measure every period's peak-to-peak values



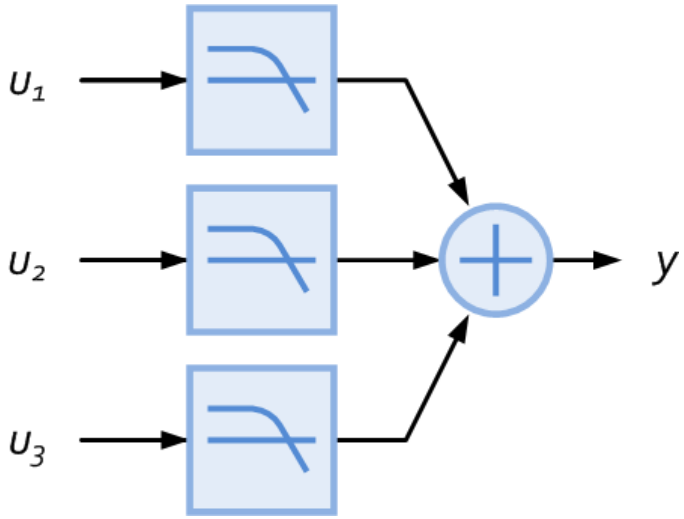
Simulated Waveforms

- Simulation results with the testbench view ***AVM_LIB.EX3:tb***



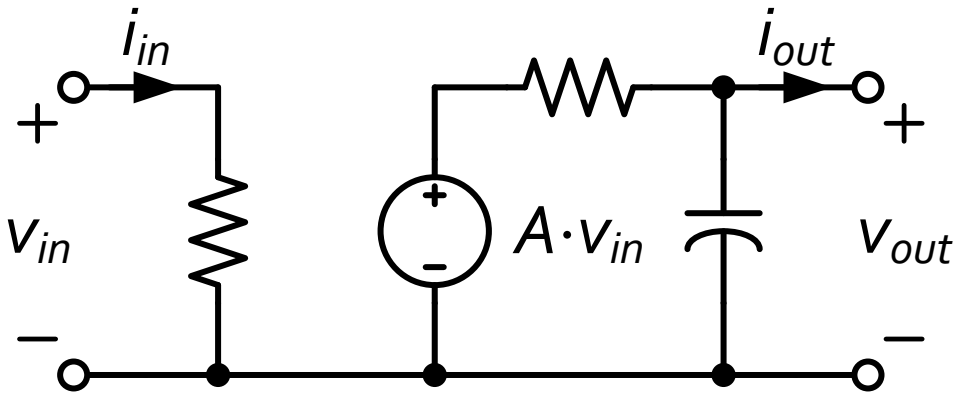
Block-Level vs. Circuit-Level Models

Block-Level Model
(Signal-flow Model)



- A network of blocks where signals flow in one direction only

Circuit-Level Model
(Conservative System Model)

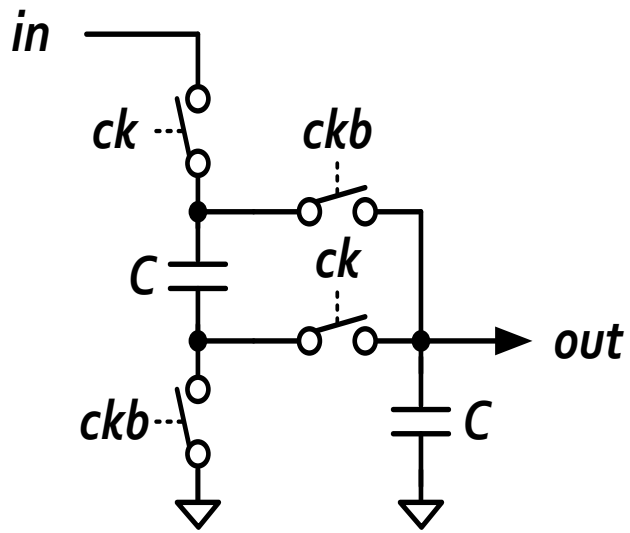


- A network of circuits whose state is described by voltages & currents
- e.g. loading effects

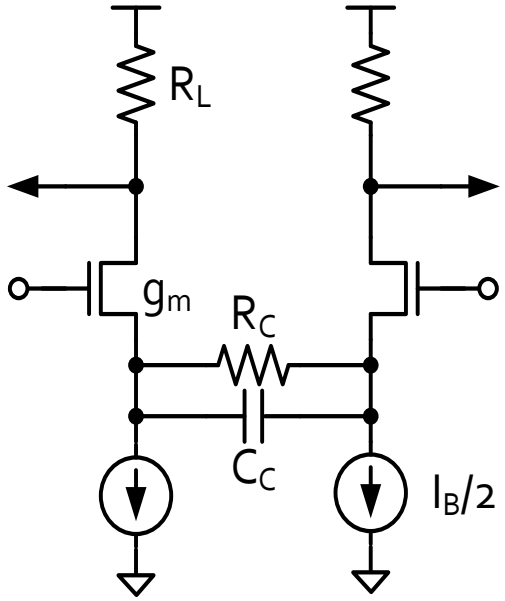
Need for Circuit-Level Models (CLMs)

- Circuit-level models are the most natural way to model switching, nonlinear, and loading effects in analog circuits

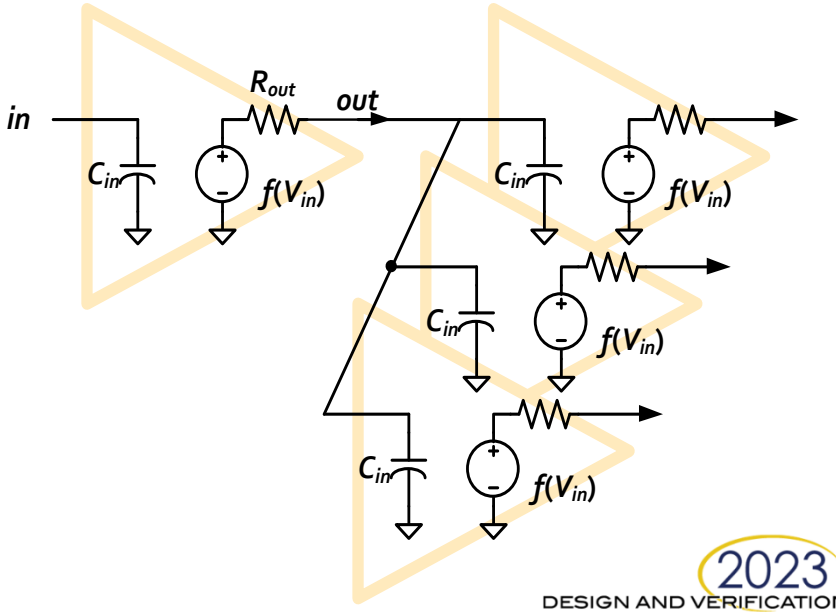
Switching Behaviors



Nonlinear Behaviors

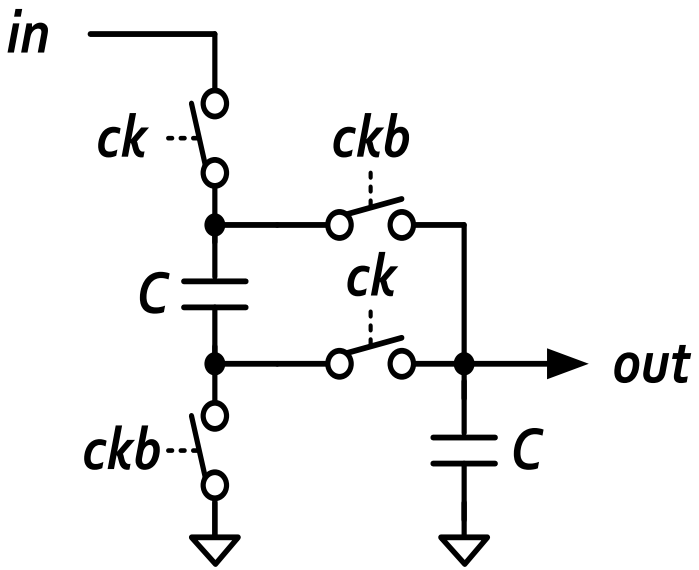


Loading Effects



CLM Support in *XMODEL*

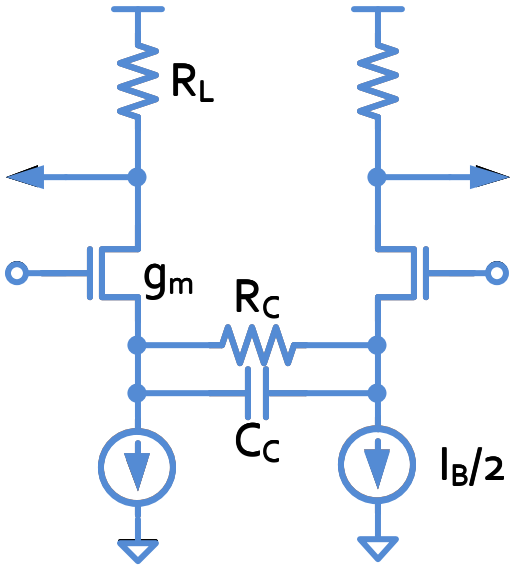
- With the *XMODEL* circuit primitives, one can describe analog circuits directly by listing their elements and devices
- *XMODEL* can simulate these models in SystemVerilog in an event-driven fashion without using SPICE




```
module sc_converter(
      input xreal in,
      output xreal out,
      input xbit ck, ckb
);

xreal       n1, n2;

switch      sw1(.pos(in), .neg(n1), .ctrl(ck));
switch      sw2(.pos(n1), .neg(out), .ctrl(ckb));
switch      sw3(.pos(n2), .neg(out), .ctrl(ck));
switch      sw4(.pos(n2), .neg(`ground), .ctrl(ckb));

capacitor   #(.C(1e-12)) C1(.pos(n1), .neg(n2));
capacitor   #(.C(1e-12)) C2(.pos(n2), .neg(`ground));

endmodule
```

Structural Model Generation

- With CLM support, one way to auto-extract models from circuits is:
  - Model each device in the circuit using the *XMODEL* circuit primitive
  - Build the circuit model by connecting the device models as in the original circuit

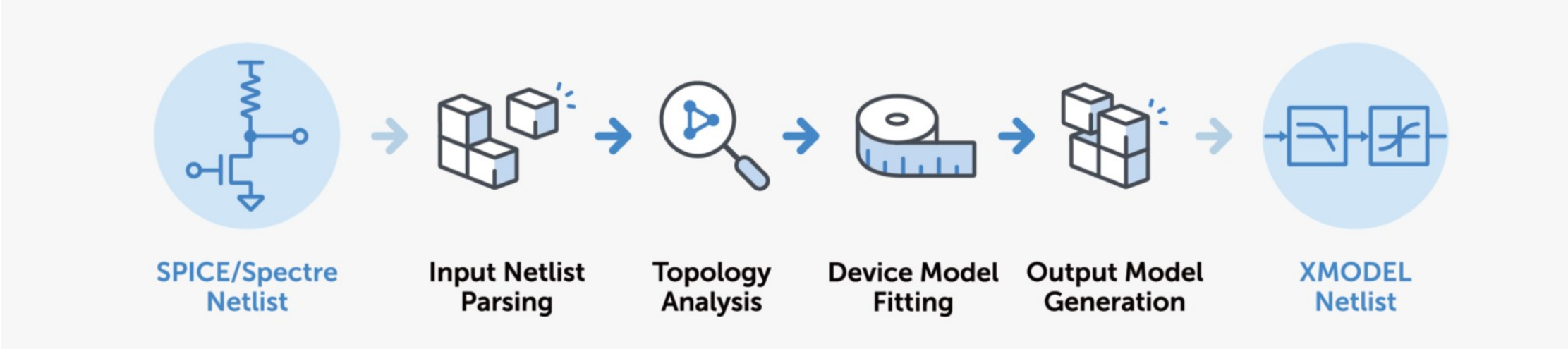


XMODEL Primitives

```
module ctle (
    `input_xreal inp, inn,          // input signals
    `output_xreal outp, outn        // output signals
);

xreal sp, sn;
xreal vdd;

vsource     #(.mode("dc"), .dc(Vdd))
            V1(.pos(vdd), .neg(`ground), .in(`ground));
isource     #(.mode("dc"), .dc(Ib/2))
            I1(.pos(sp), .neg(`ground), .in(`ground));
            I2(.pos(sn), .neg(`ground), .in(`ground));
nmosfet      #(.Kp(Gm), .Vth(Vth))
            M1(.d(outn), .g(inp), .s(sp), .b(`ground)),
            M2(.d(outp), .g(inn), .s(sn), .b(`ground));
resistor    #(.R(Rload))
            RL1(.pos(vdd), .neg(outp)),
            RL2(.pos(vdd), .neg(outn));
capacitor   #(.C(Cload))
            CL1(.pos(vdd), .neg(outp)),
            CL2(.pos(vdd), .neg(outn));
resistor    #(.R(Rc))    RC1(.pos(sp), .neg(sn));
capacitor   #(.C(Cc))    CC1(.pos(sp), .neg(sn));

endmodule
```

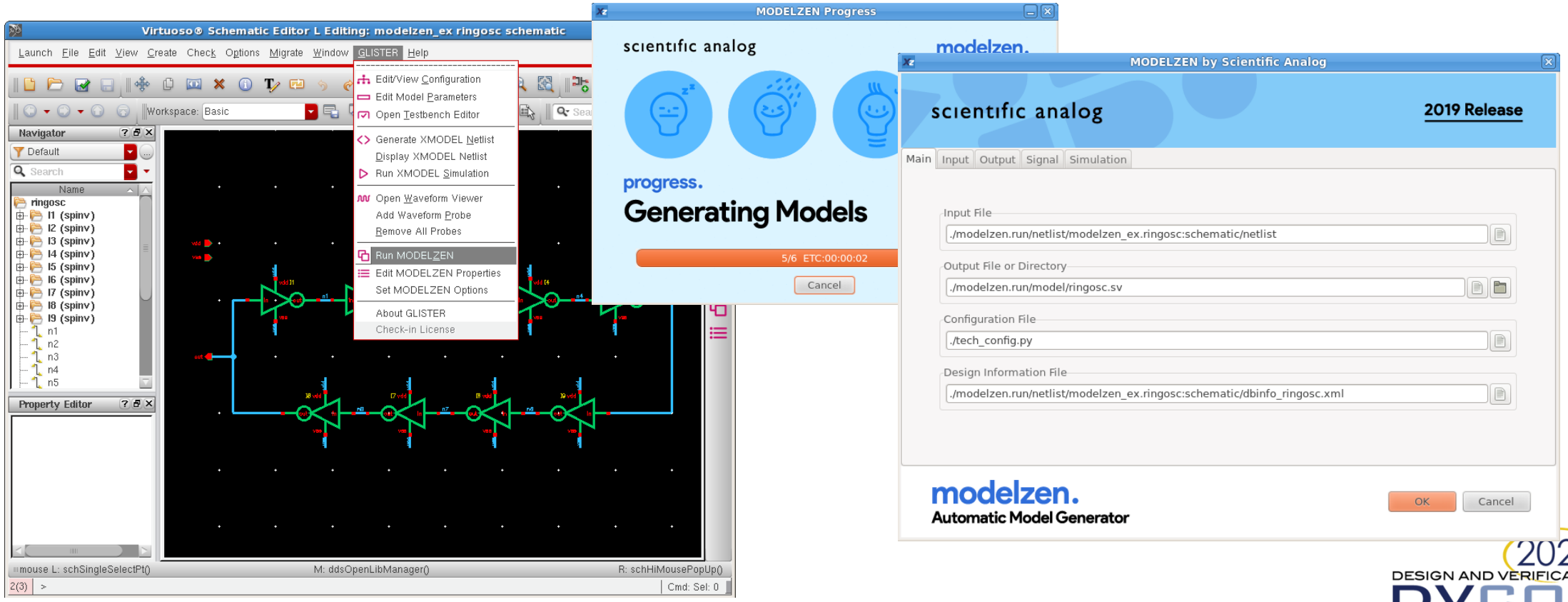
MODELZEN: Auto-Extract Bottom-Up Models

- ***MODELZEN*** can auto-extract bottom-up models from analog circuits
  - Can extract both circuit-level models & functional models
  - Model parameters are calibrated via SPICE simulations
  - Extracted models also simulate in an event-driven way



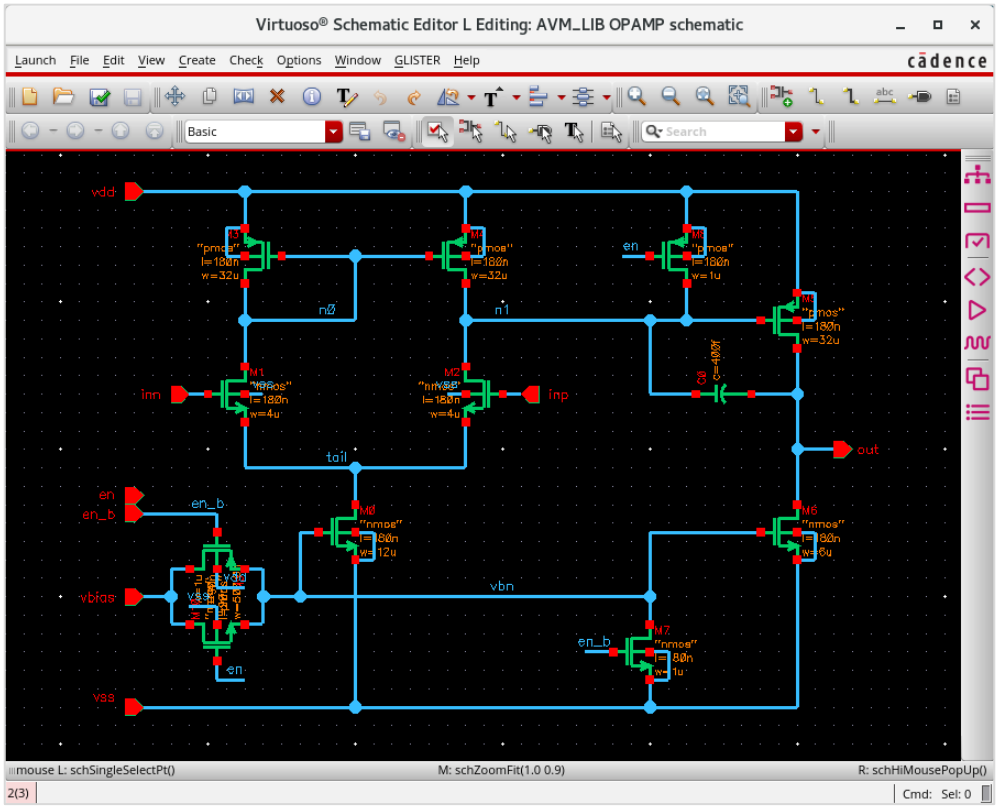
MODELZEN with *GLISTER*

- With *MODELZEN & GLISTER*, you can auto-create analog models from circuit schematics just with a single mouse click!



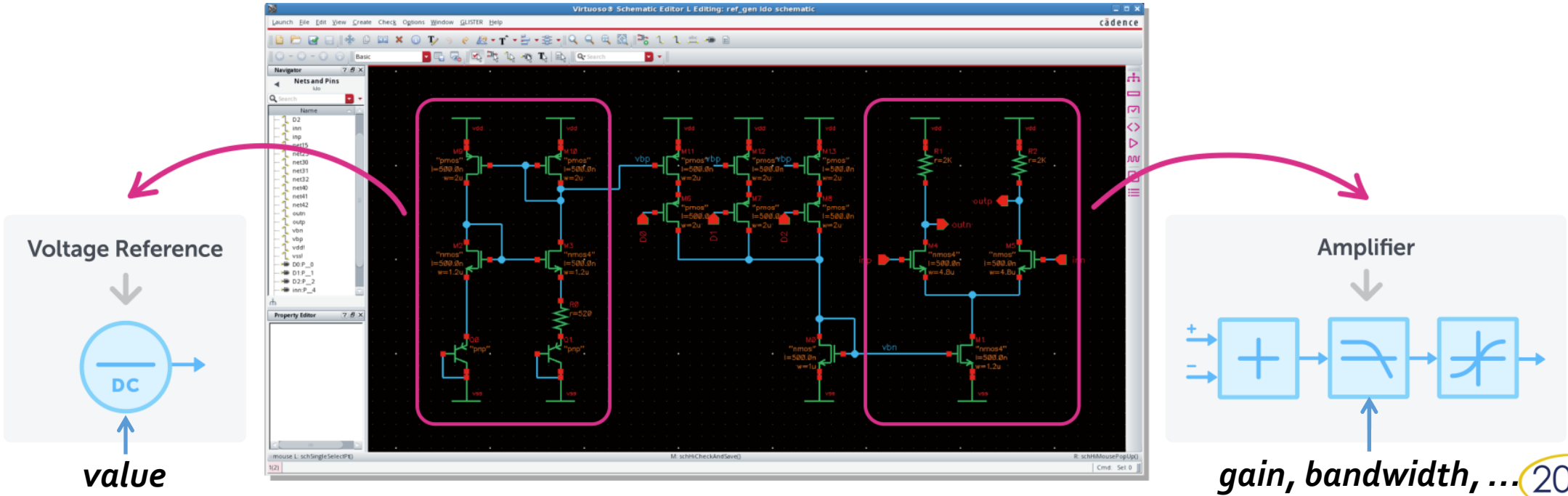
Two-Stage Op Amp Example

- Structural models can be constructed correctly without requiring analog expertise, but have limited simulation speeds due to their complexity



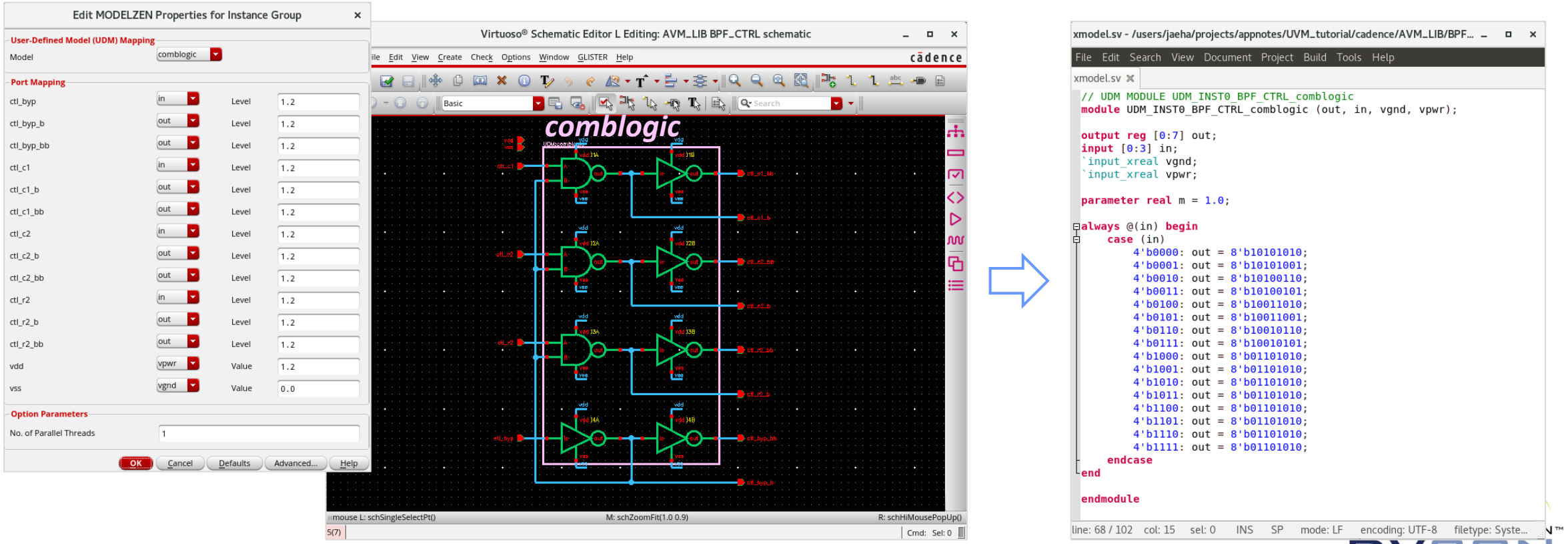
Functional Model Generation with UDM

- ***User-Defined Model*** **(*UDM*)** interface of *MODELZEN* lets you generate higher-abstraction models (e.g. functional models) for selected parts of the circuits



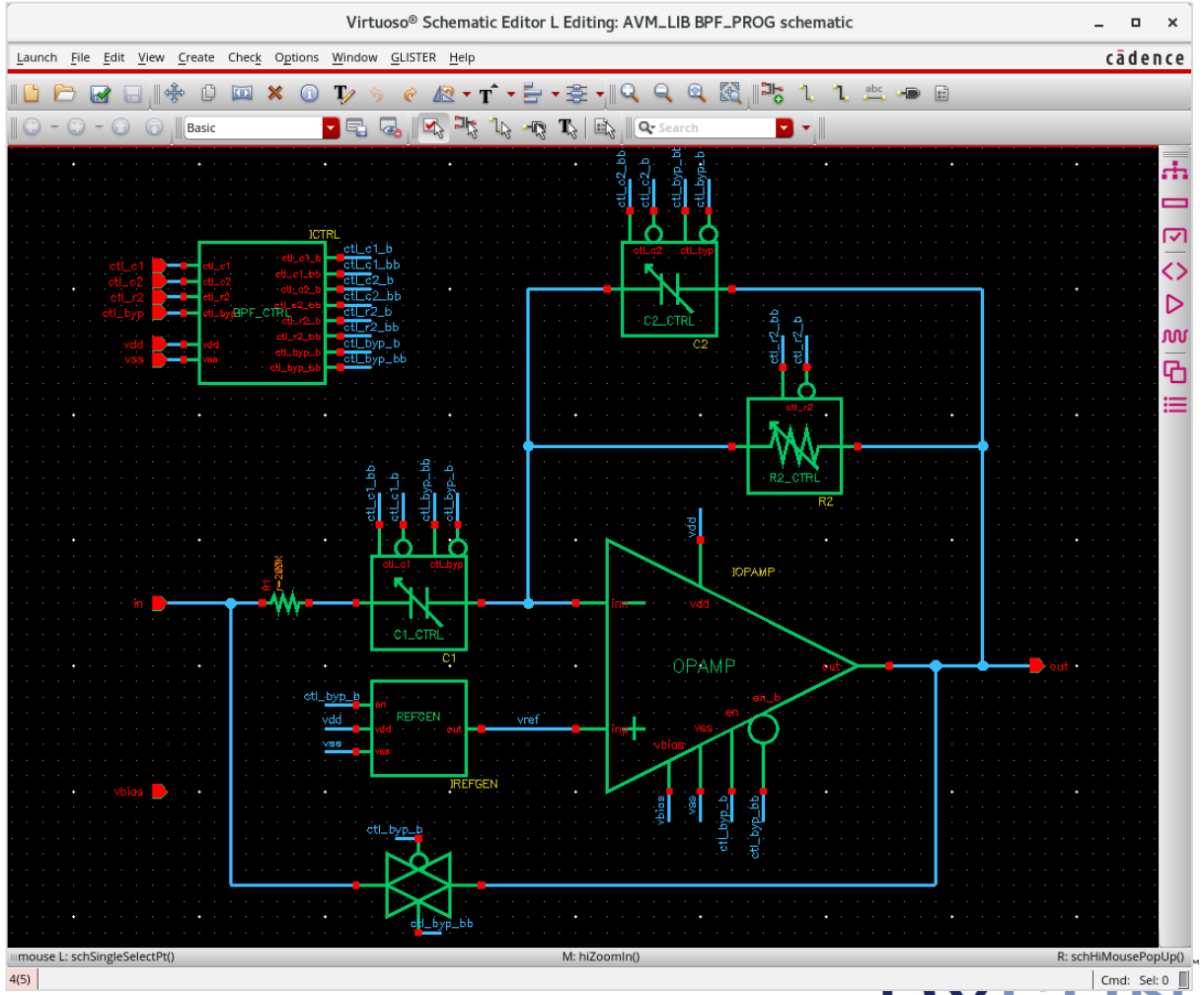
Control Logic Block Example

- *MODELZEN* generates a functional model for a circuit identified as a combinational logic (*comblogic* UDM)



Lab Exercise #4-1

- Generate SV model from a programmable bandpass filter example using MODELZEN
  - ***AVM_LIB.BPF_PROG:schematic***
- Examine the generated model stored as the cell's ***xmodel*** view
  - ***AVM_LIB.BPF_PROG:xmodel***



Generated SystemVerilog Model

- ***AVM_LIB.BPF_PROG :xmodel***

- The model preserves the hierarchy of the original circuit design

xmodel.sv - /users/jaeha/projects/appnotes/UVM_tutorial/cadence/AVM_LIB/BPF_PROG/xmodel - Geany (new instance)

File Edit Search View Document Project Build Tools Help

xmodel.sv

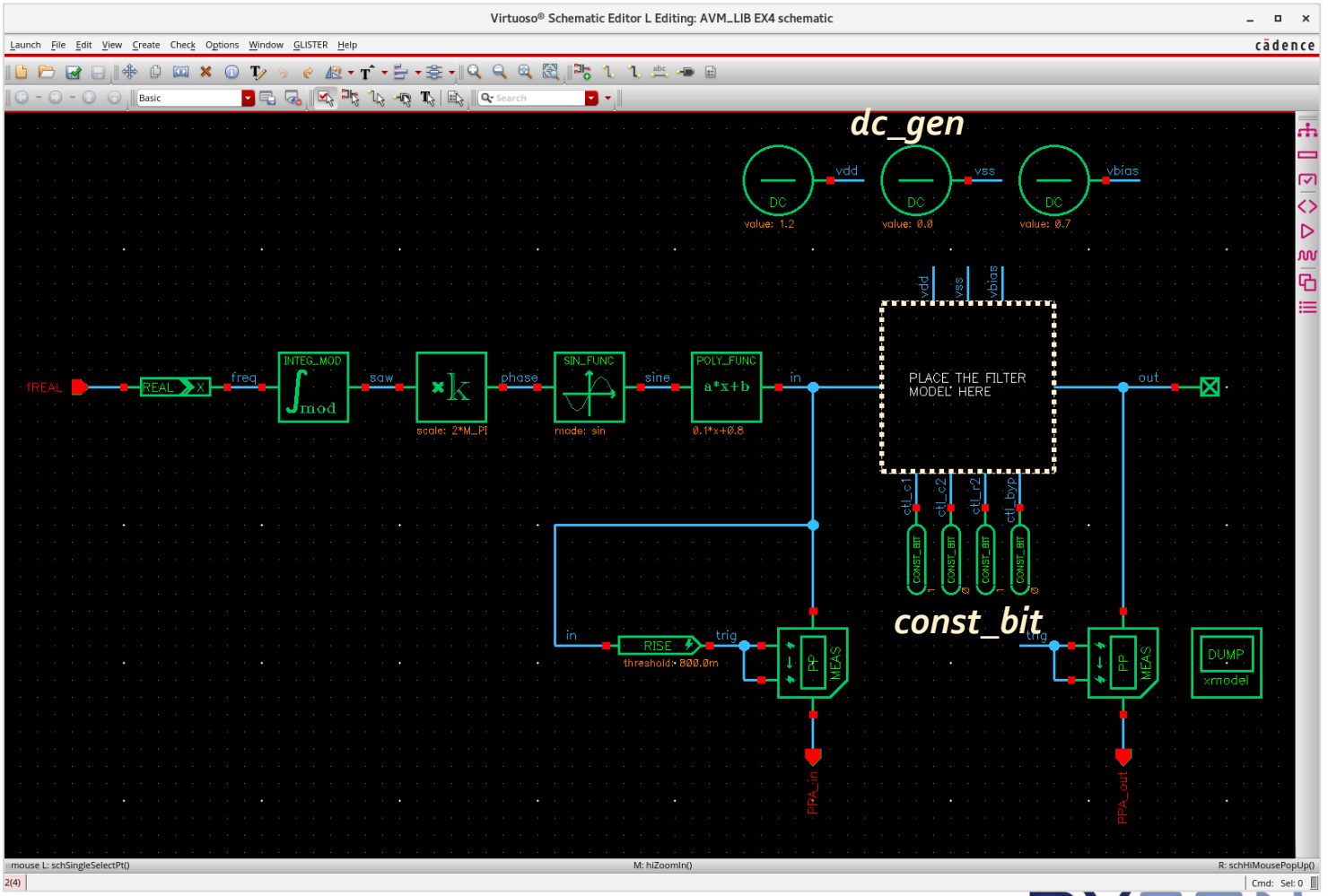
```
// XMODEL/SystemVerilog model generated from ./modelzen.run/netlist/AVM_LIB.BPF_PROG:schematic/netlist
// By MODELZEN (XMODEL Release 2022.11 (x86_64)) on Tue Jan 10 14:12:42 2023

`include "xmodel.h"

// TOP-LEVEL MODULE BPF_PROG
module BPF_PROG (out, ctl_byp, ctl_c1, ctl_c2, ctl_r2, in, vbias, vdd, vss);

`input_xreal vss;
input ctl_c1;
input ctl_c2;
`input_xreal in;
input ctl_r2;
`input_xreal out;
input ctl_byp;
`input_xreal vbias;
`input_xreal vdd;

parameter real m = 1.0;

wire ctl_byp_b;
wire ctl_byp_bb;
wire ctl_c1_b;
wire ctl_c1_bb;
wire ctl_c2_b;
wire ctl_c2_bb;
wire ctl_r2_b;
wire ctl_r2_bb;
xreal net1;
xreal net2;
xreal vref;
xreal CONN_XREAL0_ctl_byp_b;
xreal CONN_XREAL0_ctl_byp_bb;

bit_to_xreal #(.level0(0.0), .level1(1.2)) CONN_0 (.in(ctl_byp_b), .out(CONN_XREAL0_ctl_byp_b));
bit_to_xreal #(.level0(0.0), .level1(1.2)) CONN_1 (.in(ctl_byp_bb), .out(CONN_XREAL0_ctl_byp_bb));

SUB_BPF_PROG_OPAMP #(.m(m)) IOPAMP (.out(out), .en(CONN_XREAL0_ctl_byp_b), .en_b(CONN_XREAL0_ctl_byp_bb), .inn(net2), .inp(vref), .vbias(vbias), .vdd(vdd), .vss(vss));
SUB_BPF_PROG_C2_CTRL #(.m(m)) C2 (.neg(out), .pos(net2), .ctl_byp(ctl_byp_bb), .ctl_byp_b(ctl_byp_b), .ctl_c2(ctl_c2_bb), .ctl_c2_b(ctl_c2_b));
SUB_BPF_PROG_C1_CTRL #(.m(m)) C1 (.neg(net2), .pos(net1), .ctl_byp(ctl_byp_bb), .ctl_byp_b(ctl_byp_b), .ctl_c1(ctl_c1_bb), .ctl_c1_b(ctl_c1_b));
SUB_BPF_PROG_REFGEN #(.m(m)) IREFGEN (.out(vref), .en(CONN_XREAL0_ctl_byp_b), .vdd(vdd), .vss(vss));
resistor #(.R(200000), .m(m)) R1 (.pos(in), .neg(net1));
SUB_BPF_PROG_R2_CTRL #(.m(m)) R2 (.neg(out), .pos(net2), .ctl_r2(ctl_r2_bb), .ctl_r2_b(ctl_r2_b));
SUB_BPF_PROG_tgate #(.m(m)) sw4 (.a(in), .b(out), .g(CONN_XREAL0_ctl_byp_bb), .gb(CONN_XREAL0_ctl_byp_b));
SUB_BPF_PROG_BPF_CTRL #(.m(m)) ICTRL (.ctl_byp_b(ctl_byp_b), .ctl_byp_bb(ctl_byp_bb), .ctl_c1_b(ctl_c1_b), .ctl_c1_bb(ctl_c1_bb), .ctl_c2_b(ctl_c2_b), .ctl_c2_bb(ctl_c2_bb),
 .ctl_r2_b(ctl_r2_b), .ctl_r2_bb(ctl_r2_bb), .ctl_byp(ctl_byp), .ctl_c1(ctl_c1), .ctl_c2(ctl_c2), .ctl_r2(ctl_r2), .vdd(vdd), .vss(vss));

endmodule
```

line: 64 / 373 col: 18 sel: 0 INS SP mode: LF encoding: UTF-8 filetype: SystemVerilog scope: unknown

Lab Exercise #4-2

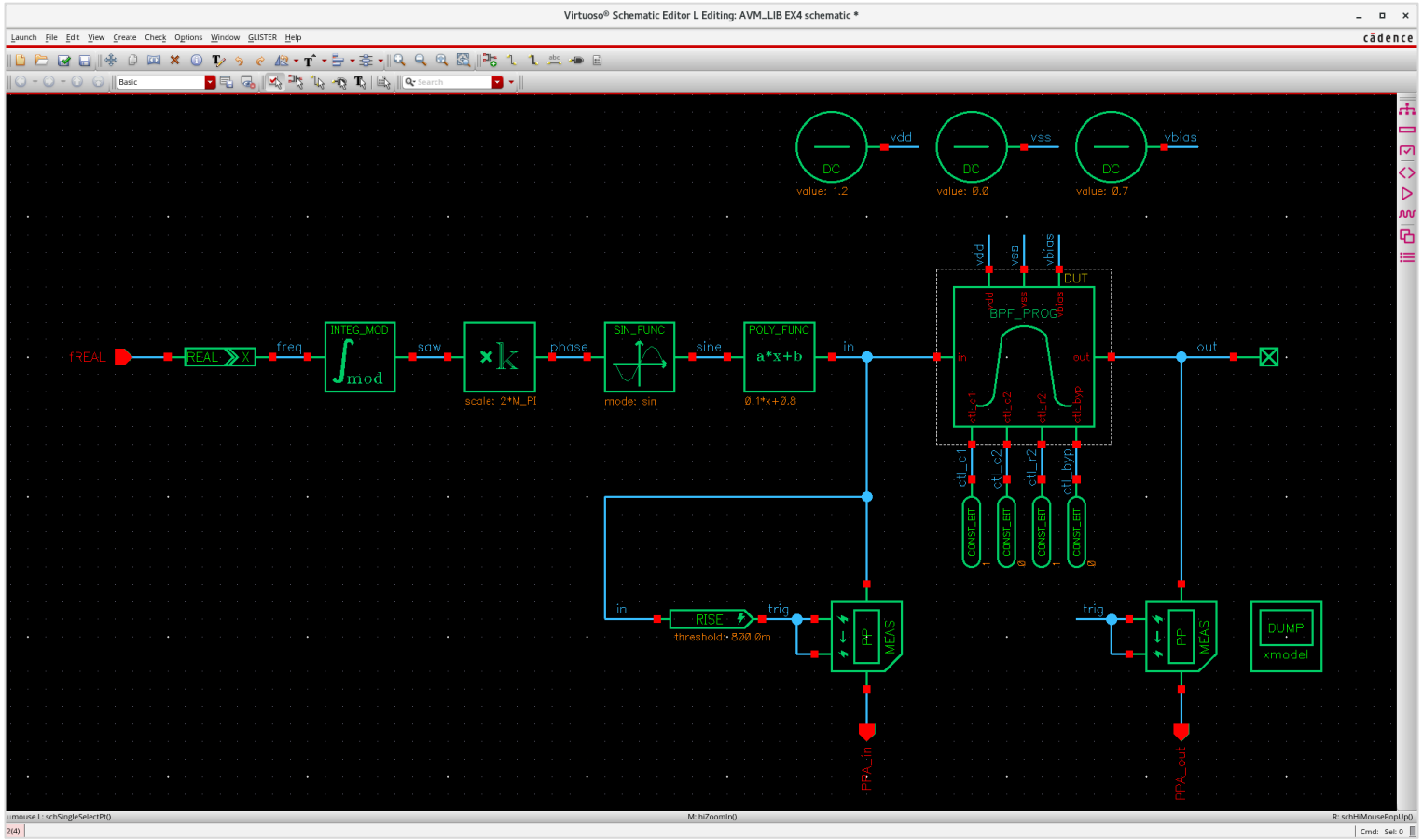
- Now place the symbol instance of ***BPF_PROG*** into our fixture and run simulation
  - ***AVM_LIB.EX4:schematic***

- This fixture also includes:
  - ***dc_gen*** primitives for supplying vdd, vss, vbias
  - ***const_bit*** primitives for setting the control bits



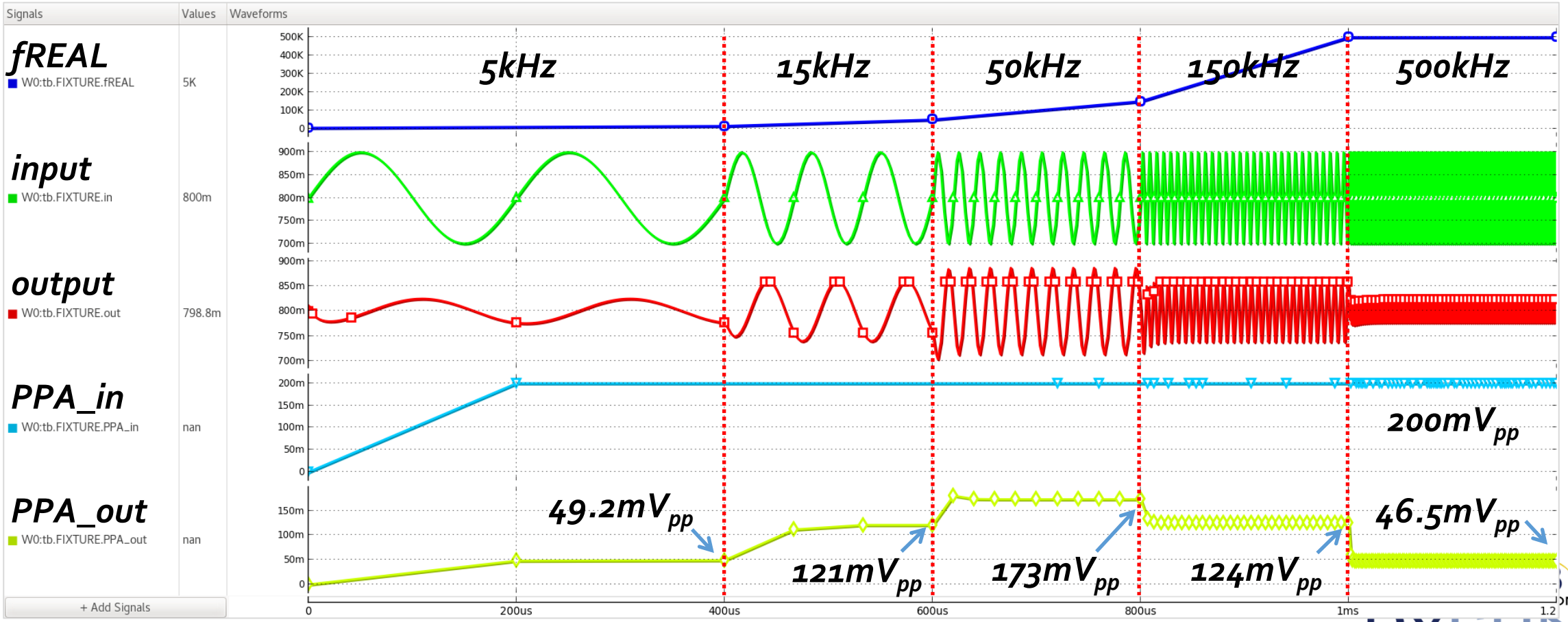
Solution for Lab Exercise #4-2

- Completed fixture schematic with the ***BPF_PROG*** cell



Simulated Waveforms

- Simulated results with the ad-hoc testbench view ***AVM_LIB.EX4:tb***



Completed Fixture Module for Analog BPF

```
module FIXTURE (
    IF_BUS FREQ_IN,
    IF_BUS AMPL_OUT
);
    xreal in, out;
    xreal vdd, vss, vbias;
    xreal freq, saw, phase, sine;
    xbit trig;

    // DUT instantiation
    BPF_PROG DUT(
        .out(out), .in(in),
        .ctl_c1(1'b1), .ctl_c2(1'b0), .ctl_r2(1'b1), .ctl_byp(1'b0),
        .vdd(vdd), .vss(vss), .vbias(vbias)
    );

    // sinusoidal input generation
    real_to_xreal    ICONN (.in(FREQ_IN.fREAL), .out(freq));
    integ_mod        IINTMOD (.in(freq), .out(saw));
    scale            #(.scale(2*M_PI)) ISCALE (.in(saw), .out(phase));
    sin_func         ISINFUNC (.in(phase), .out(sine));
    poly_func        #(.data('{0.8,0.1})) IPOLYFUNC (.in(sine), .out(in));

    // peak-to-peak amplitude measurements
    trig_rise        #(.threshold(0.8)) ITRIGRISE (.in(in), .out(trig));
    meas_pp IMEASPPO (.out(AMPL_OUT.PPA_OUT), .in(out), .from(trig), .to(trig));
    meas_pp IMEASPPI (.out(AMPL_OUT.PPA_IN), .in(in), .from(trig), .to(trig));
```

Interface busses connecting to driver/monitor (more later)

- A source file located in: ***UVM_TB/FIXTURE.sv***

Instantiation of the BPF DUT model (Lab #4)

Generation of variable-frequency sinusoidal input (Lab #2)

Measurement of input & output peak-to-peak amplitudes (Lab #3)

Completed Fixture Module for Analog BPF (2)

UVM_TB/FIXTURE.sv (cont'd)

```
    // power & bias supplies
    dc_gen #(.value(1.2)) IVDD (.out(vdd));
    dc_gen #(.value(0.0)) IVSS (.out(vss));
    dc_gen #(.value(0.7)) IVBIAS (.out(vbias));

    // generate trigger marking the completion of measurement
    bit TICK, TOCK=0;
    xbit_to_bit     ICONN_TRIG (.in(trig), .out(TICK));

    initial begin: MEASURE
        wait(!FREQ_IN.RST);
        @(posedge FREQ_IN.PKT_CLK);
        forever begin
            @(negedge FREQ_IN.PKT_CLK);
            repeat (5) @(TICK);
            TOCK = ~TOCK;
        end
    end: MEASURE

    // feeding test-management signals through
    assign AMPL_OUT.TOCK = TOCK;
    assign AMPL_OUT.TAG  = FREQ_IN.TAG;

endmodule: FIXTURE
```

Forwarding the TAG value

- Added parts for test sequencing:
  - ***fREAL*** value is updated with the positive edge of ***PKT_CLK***
  - The negative edge of ***PKT_CLK*** initiates the measurement
  - The completion trigger ***TOCK*** is toggled when the measurement trigger ***trig*** (=***TICK***) is toggled 5 times
  - The monitor component samples the ***PPA_IN*** & ***PPA_OUT*** values when ***TOCK*** is toggled (more later)

Part I: Summary

- We learned how to build a fixture module that includes:
  - AMS device under verification (DUV) modeled in SystemVerilog; and
  - Analog instrumentations for generating stimuli and measuring responses

- To do so, we used:
  - *XMODEL* primitives in *GLISTER* to compose the analog instrumentations
  - *MODELZEN* to auto-extract SV models from the analog circuits

- Next step is to build a UVM testbench around this fixture module!

Part II. Building UVM Testbench for AMS Circuits

Charles Dančak

scientific analog

Part II: Contents

§1. Key UVM Features

- UVM testbench organization
- Functions of UVM objects
- A typical TLM path for packets
- Virtual-to-physical bus "bridge"
- A phased UVM simulation
- A configurable machine

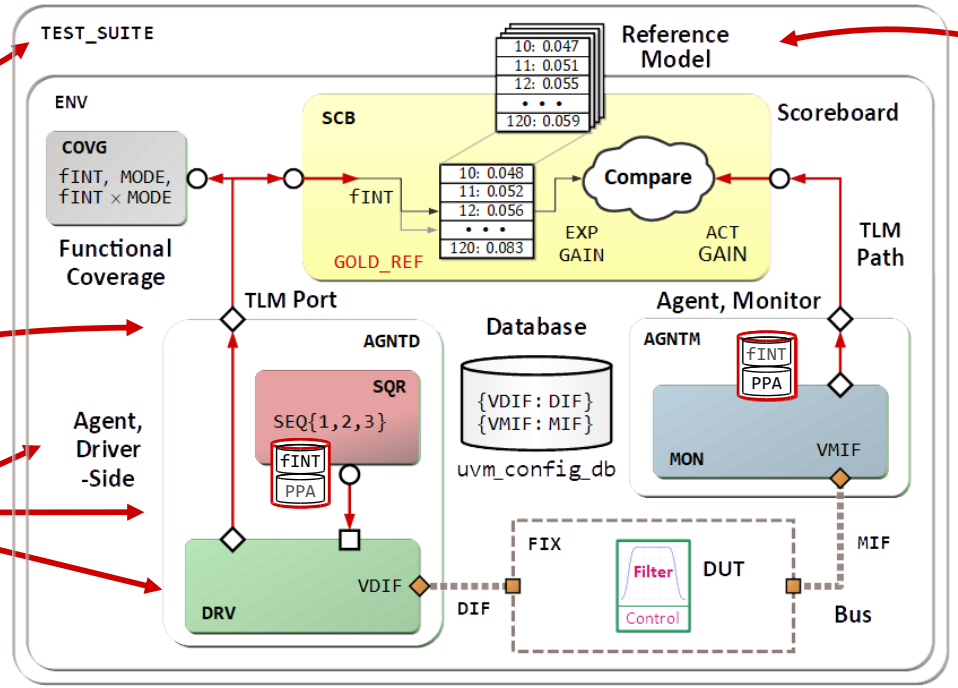
Note: Text in angle brackets (« ») indicates noncritical code whose details are omitted to avoid cluttered slides.

UVM Testbench Organization

Test-Suite Object:
Select at Run Time

TLM Path
(Broadcast)

Reusable Class-Based Hierarchy



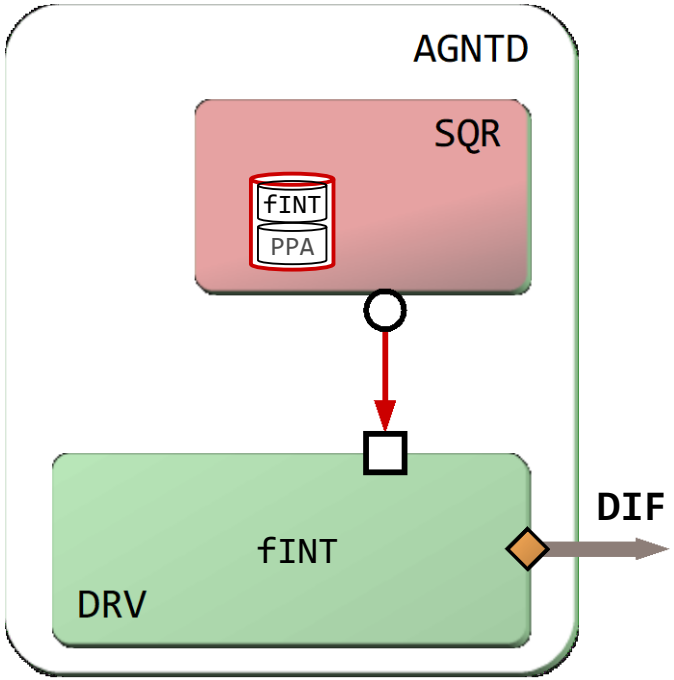
SPICE-Generated Reference Model

Module FIXTURE with DUT

- The analog filter DUT is instantiated in a fixture module.
- Virtual interfaces DIF and MIF connect it to component hierarchy.
- Data packets travel between the components via TLM paths.

Functions of UVM Objects (1/2)

- Driver applies stimulus to DUT, over the interface bus DIF. It is aware of DUT timing requirements.
- Packets carry transaction (stimulus + response) data between the component TLM ports.
- A sequencer supplies driver with a sequence of transactions that contain untimed stimulus data.
- The driver-side agent is a container class that instantiates DRV and SQR, connecting their ports.



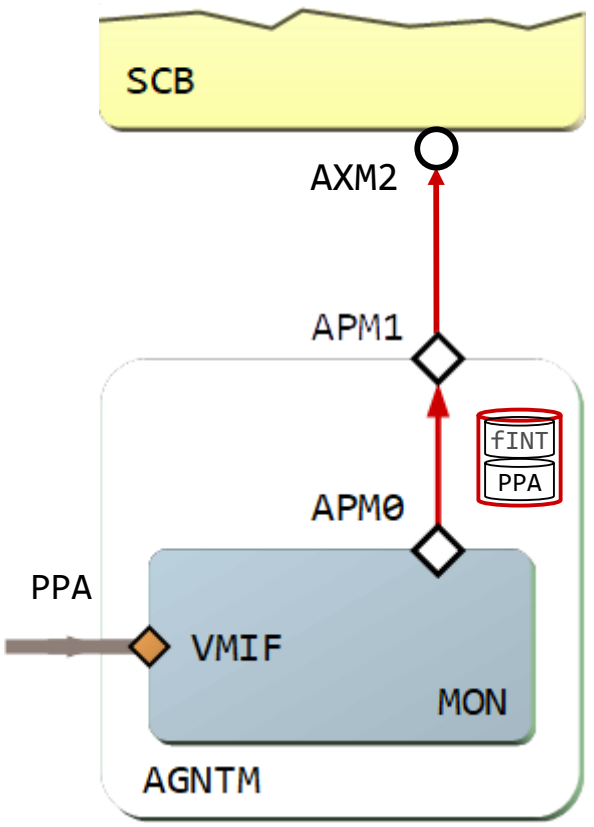
Driver-Side Agent

Functions of UVM Objects (2/2)

- Monitor gets the DUT response via interface bus `MIF`, at appropriate time in each cycle. It then assembles this data into a packet and forwards it up to the scoreboard.
- Monitor-side agent is a container that instantiates `MON`, connecting its `APM0` port up to higher levels.
- Scoreboard collects stimulus and response packets, then evaluates the results of each transaction.

```
//Gain discrepancy for packet N:
  gERROR[N] = gACTUAL - gEXPECT;
```

Scoreboard stores the errors into an array

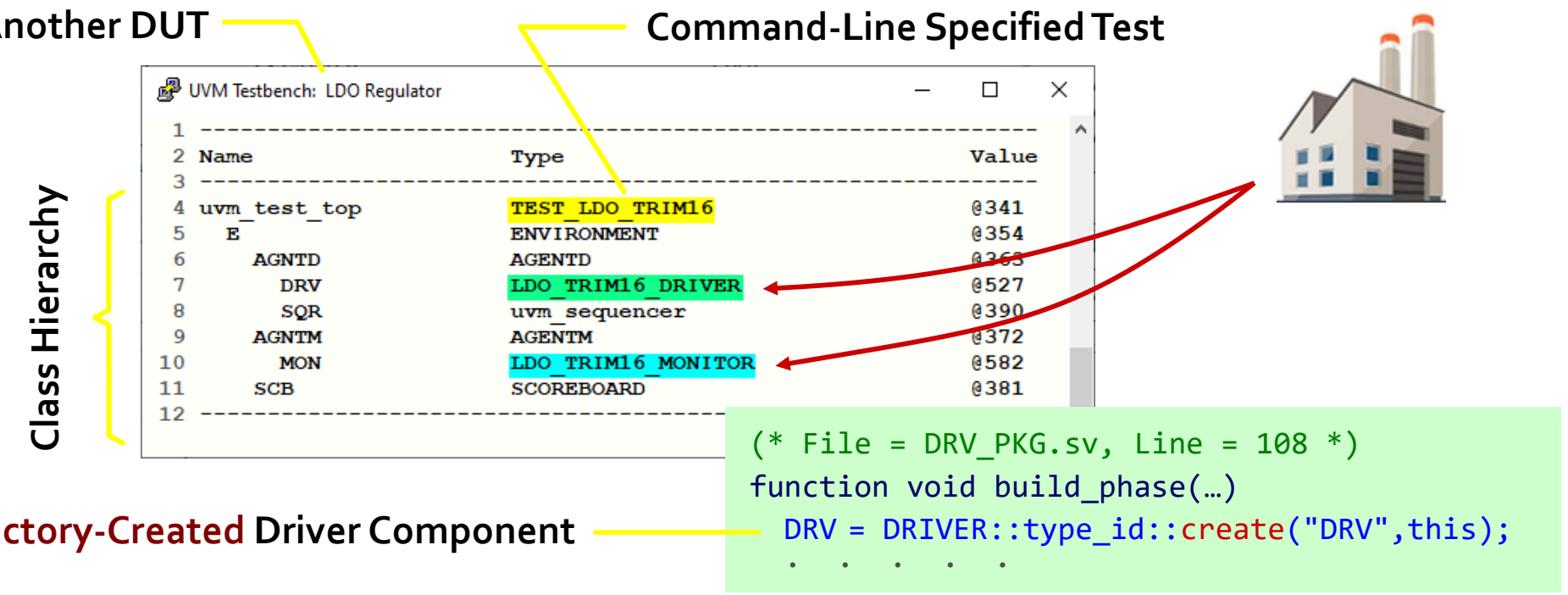


Monitor-Side Agent

Construct vs. Create

Another DUT

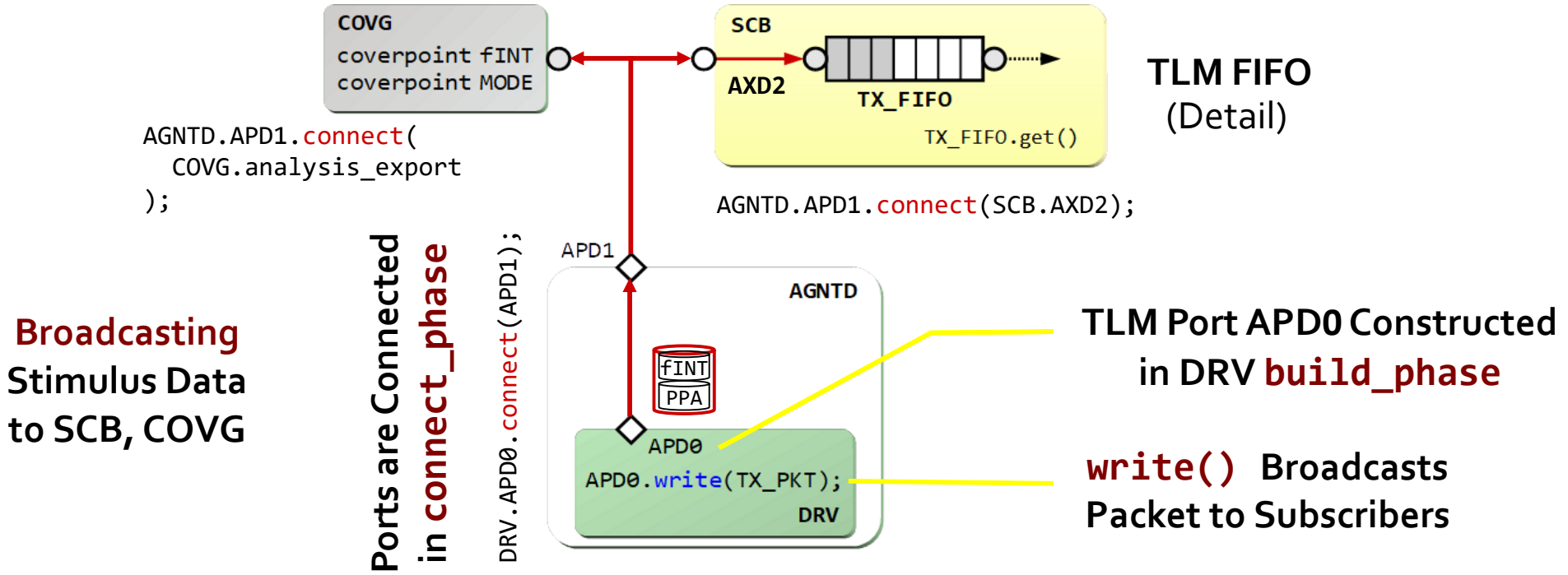
Command-Line Specified Test



Factory-Created Driver Component

- In OOP testbenches, class objects are constructed via new().
- UVM lets you create() an object through a factory mechanism.
- Then one type of driver can be substituted for another, etc.

A Typical TLM Pathway



- An OOP testbench conveys packets through a FIFO `mailbox` object.
- UVM uses a mailbox abstraction: transaction-level TLM1 pathways.
- Each pathway is built by connecting its TLM ports together.

Virtual to Physical Bus (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 virtual VMIF to physical MIF.
- UVM instead stores physical bus instance MIF in a decentralized database.
- What is stored is effectively a pointer to the physical bus instance.
- The value stored under name Key_VMIF is then retrieved by MON.

Virtual to Physical Bus (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 static method get() to retrieve value under field name.
- Pointer to bus instance MIF known as a virtual interface.
- VMIF is now a class variable pointing to a physical 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

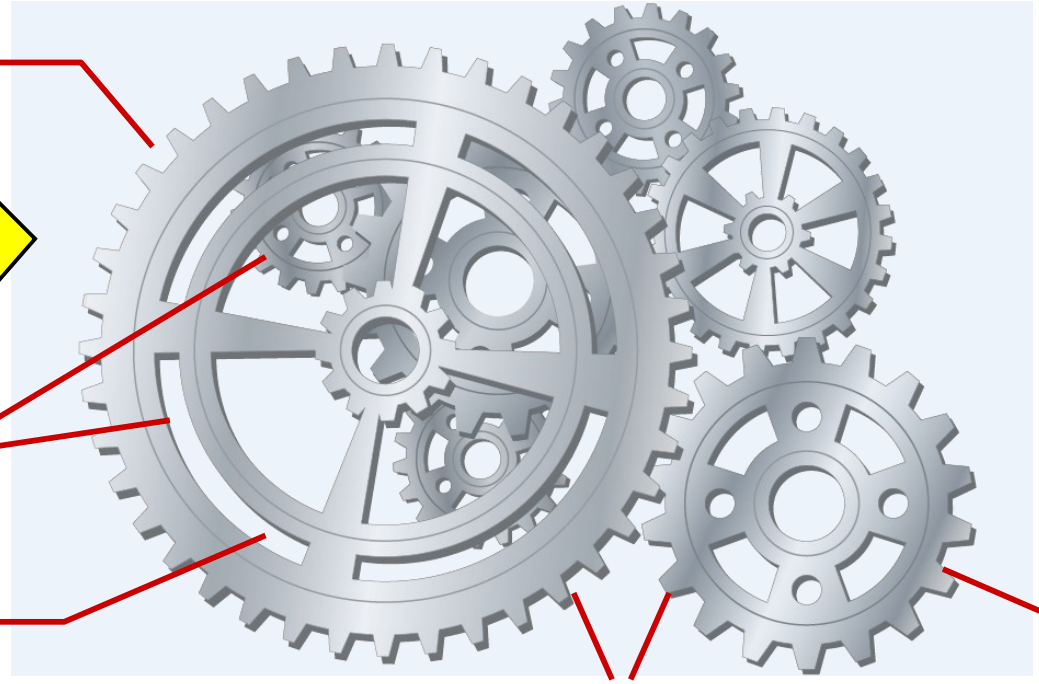
At Time 0.00:
- Component classes built.
- Configuration `set/get()`.
- TLM ports interconnected.

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

A Configurable Machine

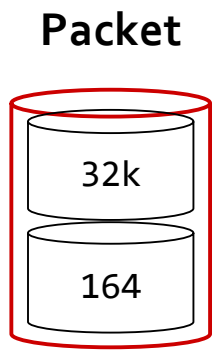


- Mechanical analogy: UVM resembles a configurable machine.
- Provides the infrastructure needed for any complex testbench.
- Utilize only the functionality you need to verify the DUT.

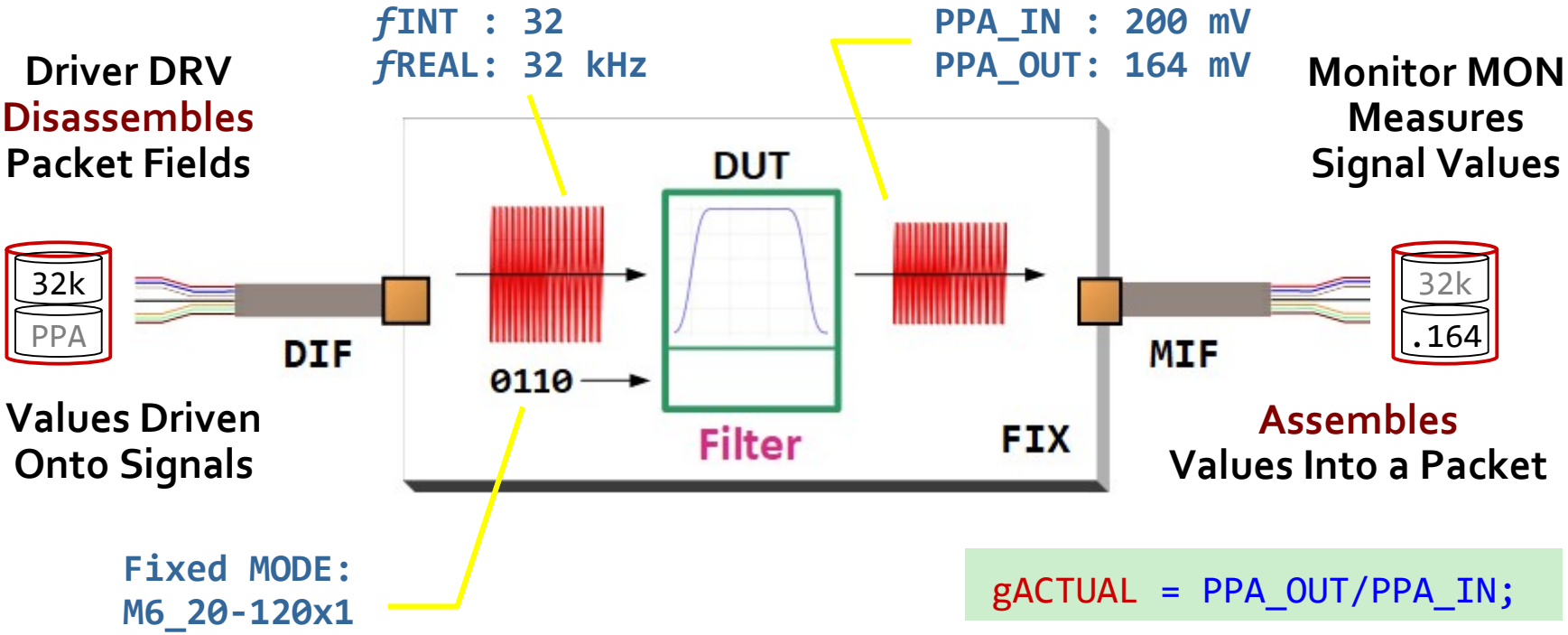
§2. A Basic Filter Test

- A basic test scenario
- Why use UVM packets?
- **Lab #5:** Code a packet

- A stock UVM sequencer
- Generate a packet stream
- Simulate the packet stream
- DUT-specific timing budget
- **Lab #6:** Code a sequence



A Basic Test Scenario



- To test a bandpass filter DUT, we want to build a UVM testbench that measures the input-to-output gain (gACTUAL) at randomized frequencies.

Why Use UVM Packets?

Typical UVM Packet

Random Variable Storing Frequency

Constrains the Range of Frequency Values

```
class PACKET extends uvm_sequence_item;
  . . . .
//Packet tag:
  int TAG = 0;

//Stimulus fields:
  rand int fINT;  //Range 5--500 kHz.
  real     fREAL; //Equivalent, in Hz.

//Constrain fINT to design range:
  constraint fRANGE_con {
    fINT inside { [5:500] };
  }

  «continued on next slide»
```

Packet Class

- Data in a packet class can be randomized and constrained.
- Using dynamic class objects avoids out-of-memory issues.
- Plain variables — array or **`struct`** — lack these features.

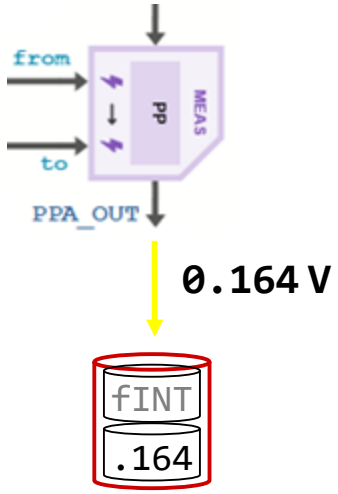
Reusing the Packet Class

Frequency is Randomized in `int` and Cast to `real` to Measure its Coverage

Constructor Must Call `super.new()`

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

//Measured amplitude:
  real PPA_IN, PPA_OUT; //V.

  function new(…);
    super.new(…);
  endfunction: new

endclass: PACKET
```

Packet Class



Measured Values from DUT are put into Packet

- The same packet type is used on the monitor side, maximizing reuse.
- Monitor reads `PPA_IN` & `PPA_OUT` from `MIF` bus, writes them into a packet.
- Packets are sent up to scoreboard to compare with reference data.

Lab #5: Code a Data Packet

(1) Insert UVM base class.

(2) Make fINT random.

(3) Constrain to audio range.

(4) Specify data type.

```
(* File = DATA_PKG.sv, Line = 32 *)
class PACKET extends «uvm_base_class»;
  . . . . .
//Stimulus field:
  «qualifier» int fINT; //5--500 kHz.
  real fREAL;           //Equivalent.

//Constrain fINT to design range:
  constraint fRANGE_con {
    fINT inside { [«range»] };
  }
  . . . . .
//Measured output (volts):
  «type» PPA_IN, PPA_OUT;

endclass: PACKET
```

Funny braces (*…*) indicate source filename and line number.

Packet Object
TX_PKT

- Replace angle-bracketed hints with SystemVerilog code
- Run **`make lab5`** to check your syntax

Lab #5: Solutions

(1) Insert UVM base class.

(2) Make fINT random.

(3) Constrain to audio range.

(4) Specify data type.

```
(* File = DATA_PKG.sv, Line = 32 *)
class PACKET extends uvm_sequence_item;
  .  .  .  .  .
//Stimulus field:
  rand int fINT;     //5--500 kHz.
  real fREAL;        //Equivalent.

//Constrain fINT to design range:
  constraint fRANGE_con {
    fINT inside { [5:500] };
  }
  .  .  .  .  .
//Measured output (volts):
  real PPA_IN, PPA_OUT;

endclass: PACKET
```

Packet Object
TX_PKT

- Each packet represents one transaction — stimulus or response.
- Field fINT will be randomized during the packet sequence.

A Stock UVM Sequencer

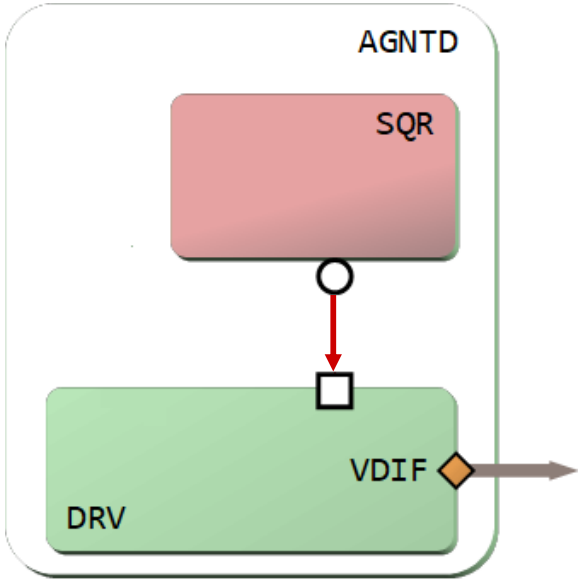
Construct, not Create

Connect TLM Ports

```
(* File = DRV_PKG.sv, Line = 85 *)
  class AGENTD extends uvm_agent;
    . . . . .
    uvm_sequencer #(PACKET) SQR;
    DRIVER  DRV;

    function void build_phase(…);
      SQR = new("SQR", this);
      . . . . .
      DRV = DRIVER…create("DRV",this);
    endfunction: build_phase

    function void connect_phase(…);
      . . . . .
      DRV.«port».connect(SQR.«export»);
    endfunction: connect_phase
  endclass: AGENTD
```

Typed Base-Class Sequencer

Driver-Side Agent



- A sequence of packets is sent to the driver by the sequencer.
- The driver declares — but does not itself create — packets.

Generate a Packet Stream

Required Name

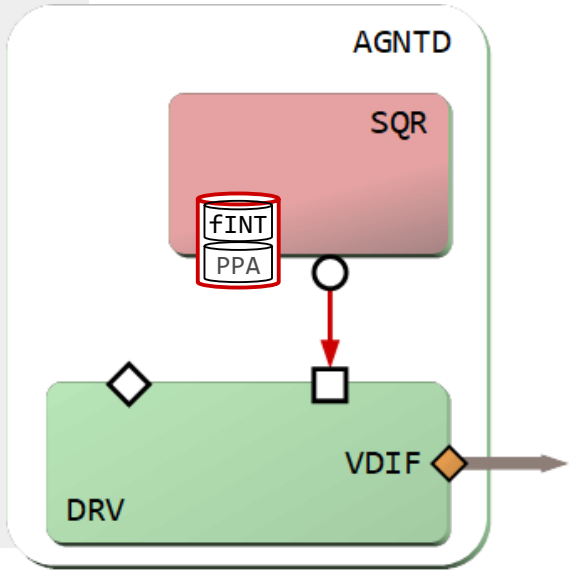
Untimed Algorithm

```
class SEQ_FILTER extends uvm_sequence #(PACKET);
  PACKET  TX_PKT; //Declare packet.
  . . . . .
  task body();  //Generate packets.
    TX_PKT = PACKET…create(…);
    . . . . .
    for («Loop for TRIALS»)
    begin:LOOP
      start_item(TX_PKT);
      ++TX_PKT.TAG; //Integer ID field.
      TX_PKT.randomize(); //fINT randomized.
      finish_item(TX_PKT);
    end:  LOOP

  endtask: body

endclass: SEQ_FILTER
```

Typical UVM Sequence

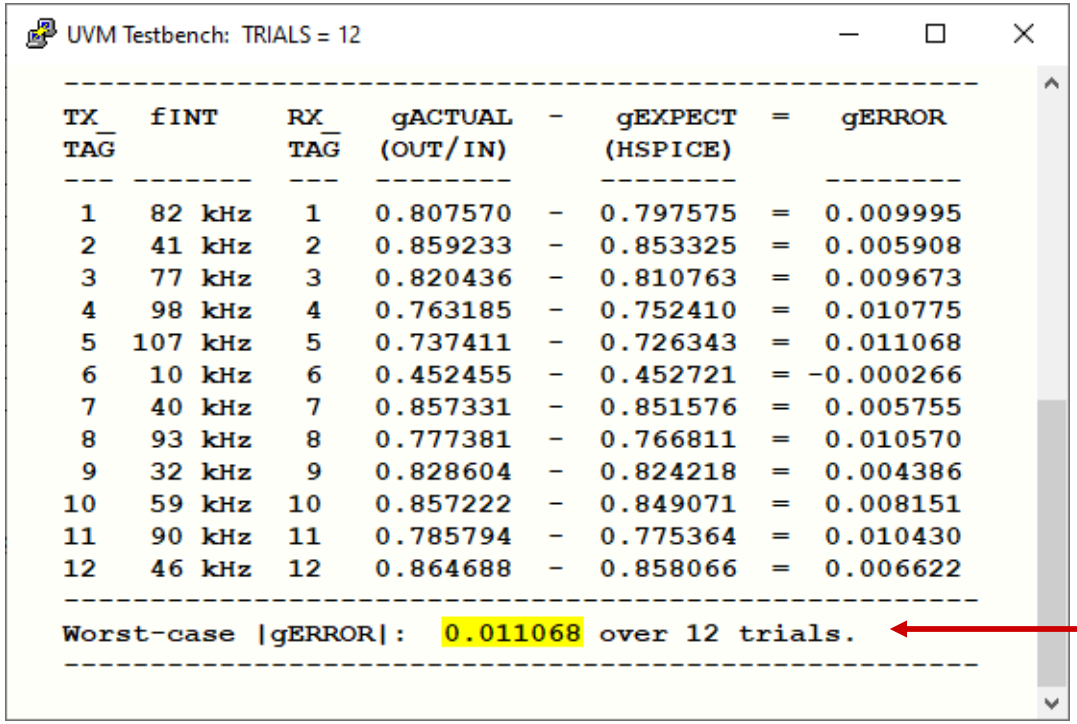
Knob

+TRIALS=3



- A sequence of packets is implemented as an untimed algorithm.
- Driver will apply fields to VDIF, until this sequence is finished.

Simulate the Packet Stream



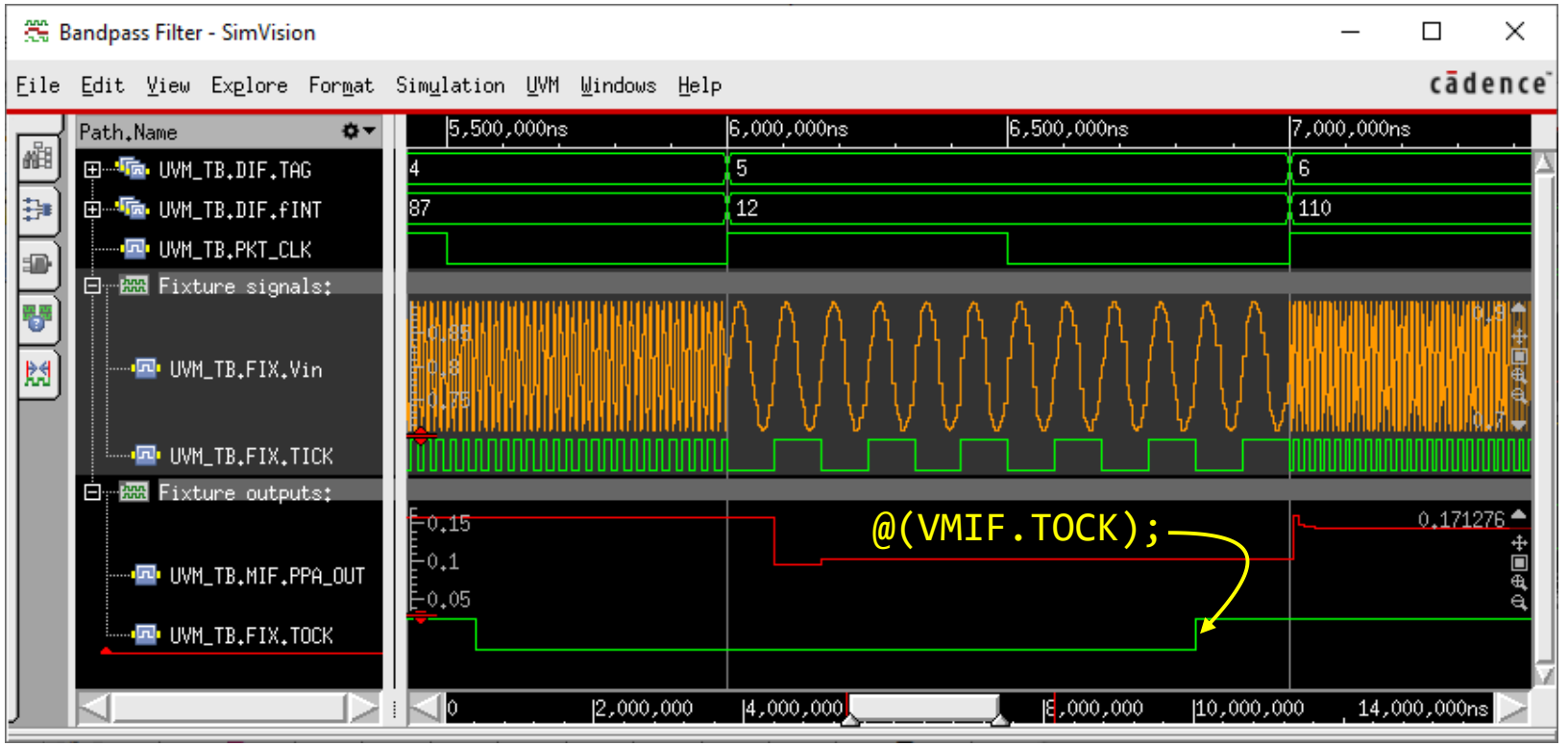
TRIALS
Iterations

Random Seed:
1279

SPICE-Like Accuracy

- Exactly when to apply the random frequency is left up to the driver.
- Similarly, when to measure PPA values is left up to the monitor.
- Notice scoreboard checks for matching TX_TAG and RX_TAG.

DUT-Specific Timing



Guideline:
Write the DRV and MON code from an explicit timing diagram.

- We apply the random frequency on active clock edges.
- And begin measuring PPA upon inactive clock edges.

Lab #6: Code a Sequence

(1) Insert type of packets.

(2) Specify for loop.

(3) Packet type.

(4) Randomize **rand** fields.

```
(* File = SEQ_PKG.sv, Line = 09 *)
class SEQ_FILTER extends uvm_sequence #(«type»);

  PACKET  TX_PKT; //Declare packet.
  . . . . .
  task body();  //Generate packets.
    TX_PKT = PACKET…create(…);
    . . . . .
    for («Loop for TRIALS»)
    begin:LOOP
      start_item(«PKT_object»);
      ++TX_PKT.TAG; //Integer ID field.
      «Randomize TX_PKT.»
      finish_item(TX_PKT);
    end:  LOOP
  endtask: body

endclass: SEQ_FILTER
```

A UVM Sequence

- Replace angle-bracketed hints with SystemVerilog code.

Lab #6: Solutions

(1) Insert type of packets.

(2) Specify for loop.

(3) Packet type.

(4) Randomize **rand** fields.

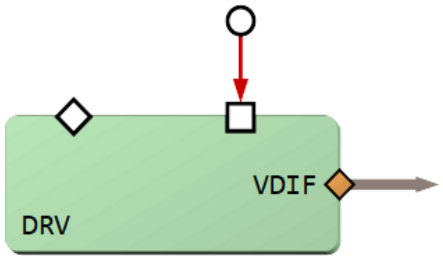
```
(* File = SEQ_PKG.sv, Line = 09 *)
class SEQ_FILTER extends uvm_sequence #(PACKET);

  PACKET  TX_PKT; //Declare packet.
  . . . . .
  task body();  //Generate packets.
    TX_PKT = PACKET…create(…);
    . . . . .
    for (int I=1; I <= TRIALS; I++)
    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
```

A UVM Sequence

- All DUT-related timing details are left up to driver and monitor.

§3. Driver and Monitor

- Typical component: driver
- Driver `run_phase()`task
- The `SEQ-DVR` handshake
- Monitor setup code
- Monitor `run_phase()`task
- Building a typical agent



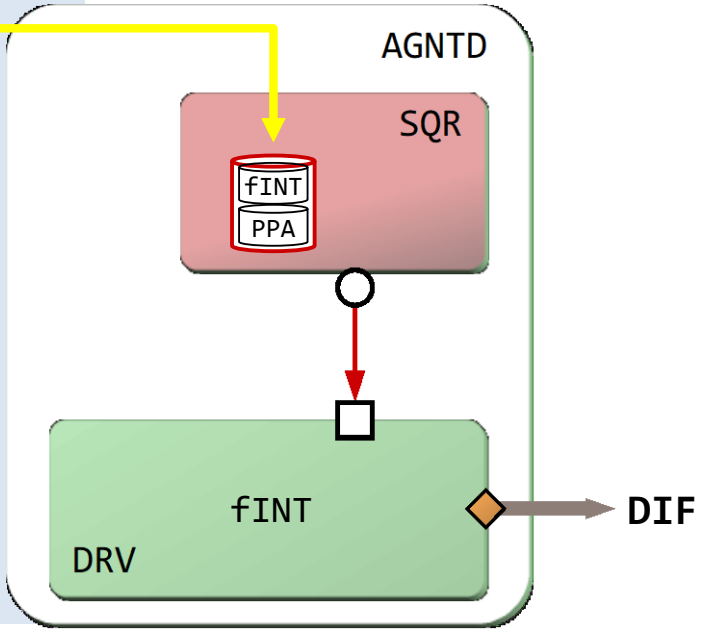
Typical Component: Driver

Set-Up Code

Task Code

```
class DRIVER extends uvm_driver #(PACKET);
  VIF_t  VDIF;   //Virtual interface.
  PACKET TX_PKT; //Declare a packet.
  . . . . .
  function void build_phase(… phase);
  //Associate instance DIF with VDIF:
    «Use uvm_config_db to get VDIF.»
    . . . .
  endfunction: build_phase

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

endclass: DRIVER
```

Driver Class

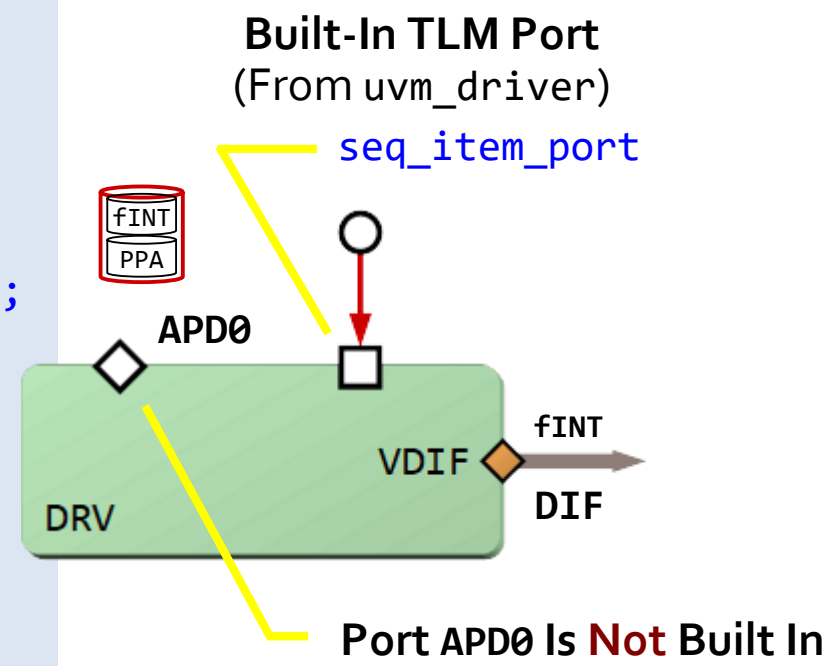


- Driver is a typical component derived from a base class.
- Does not create a packet—but pulls it down from SQR.
- Extracts fINT from each packet and drives it into the DUT.

Driver Task: run_phase()

```
(* File = DRV_PKG.sv, Line = 48 *)
task run_phase(uvm_phase phase);
  wait(!VDIF.RST);
  forever
  begin:LOOP //Drive at active edges:
    @(posedge VDIF.PKT_CLK);
    seq_item_port.get_next_item(TX_PKT);
  //Bus Signal ← Packet Field
    VDIF.TAG   = TX_PKT.TAG;
    VDIF.fINT  = TX_PKT.fINT;
     .  .  .  .  .
    seq_item_port.item_done(); ...
  end:  LOOP
  .  .  .  .  .
endtask: run_phase
```

Driver Task



- Each component's run_phase() task runs concurrently.
- Every loop iteration begins at active edge of PKT_CLK.
- Same packet is broadcast, via APD0, to SCB and COVG.

The SEQ-DRV Handshake

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

```
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

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

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

Driver

TX_PKT Argument

Apply Stimulus

Argument Optional

(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.

- Four `*item*` tasks do a full sequence-driver handshake.

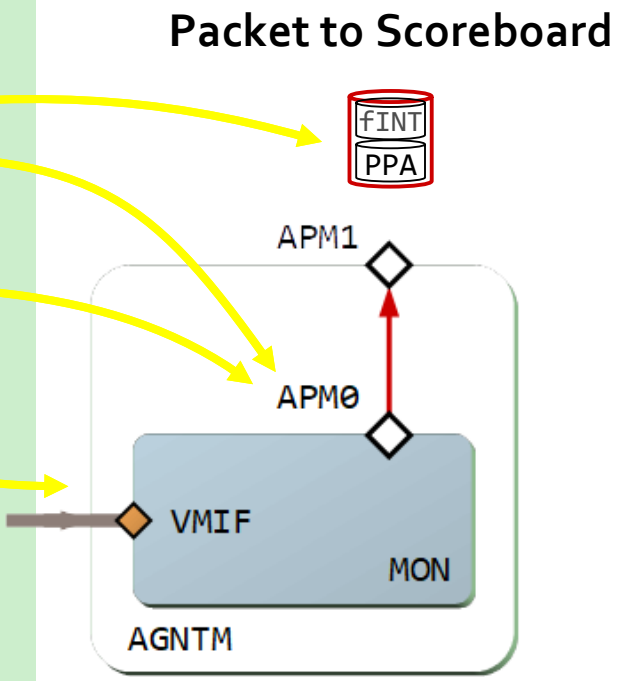
Monitor Setup Code

Set-Up Code

Task Code

```
class MONITOR extends uvm_monitor;
//Get PPA_OUT off bus; put in packet:
  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 Class



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

Monitor run_phase() Task

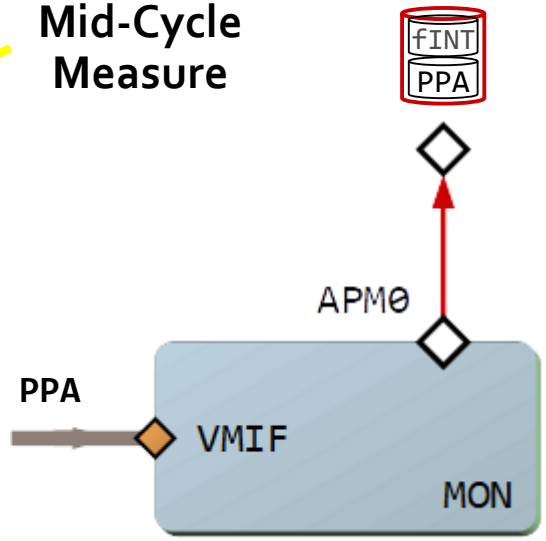
Task Code

```
(* File = MON_PKG.sv, Line = 58 *)
task run_phase(uvm_phase phase);
  wait(!VMIF.RST);
  forever
  begin:LOOP //Sample at 4  tick:
    @(VMIF.TOCK);
    RX_PKT = PACKET…create("RX_PKT");
  //Packet Fields ← Bus Signals
    RX_PKT.TAG     = VMIF.TAG;
    RX_PKT.PPA_OUT = VMIF.PPA_OUT;
    . . . . .
    APM0.write(RX_PKT); //Send up.
    . . . . .
  end:  LOOP
endtask: run_phase
```

Monitor Task

write() to Scoreboard

Mid-Cycle Measure



- Monitor assembles packet RX_PKT from the signals on VMIF.
- Actual PPA is assigned around mid-cycle to a packet field.
- Method write() sends up the filled packet to scoreboard.

Building a Typical Agent

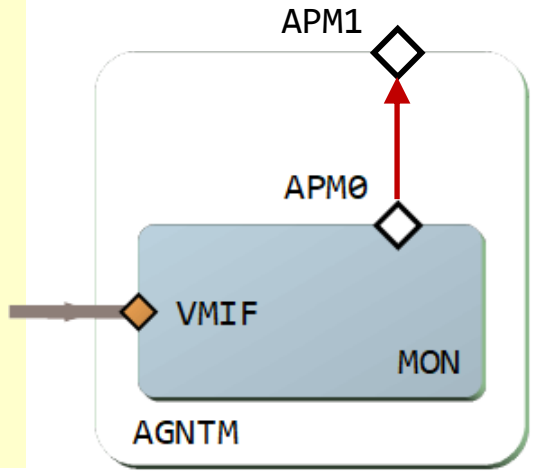
Construct

Factory-Create

```
(* File = MON_PKG.sv, Line = 93 *)
  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.



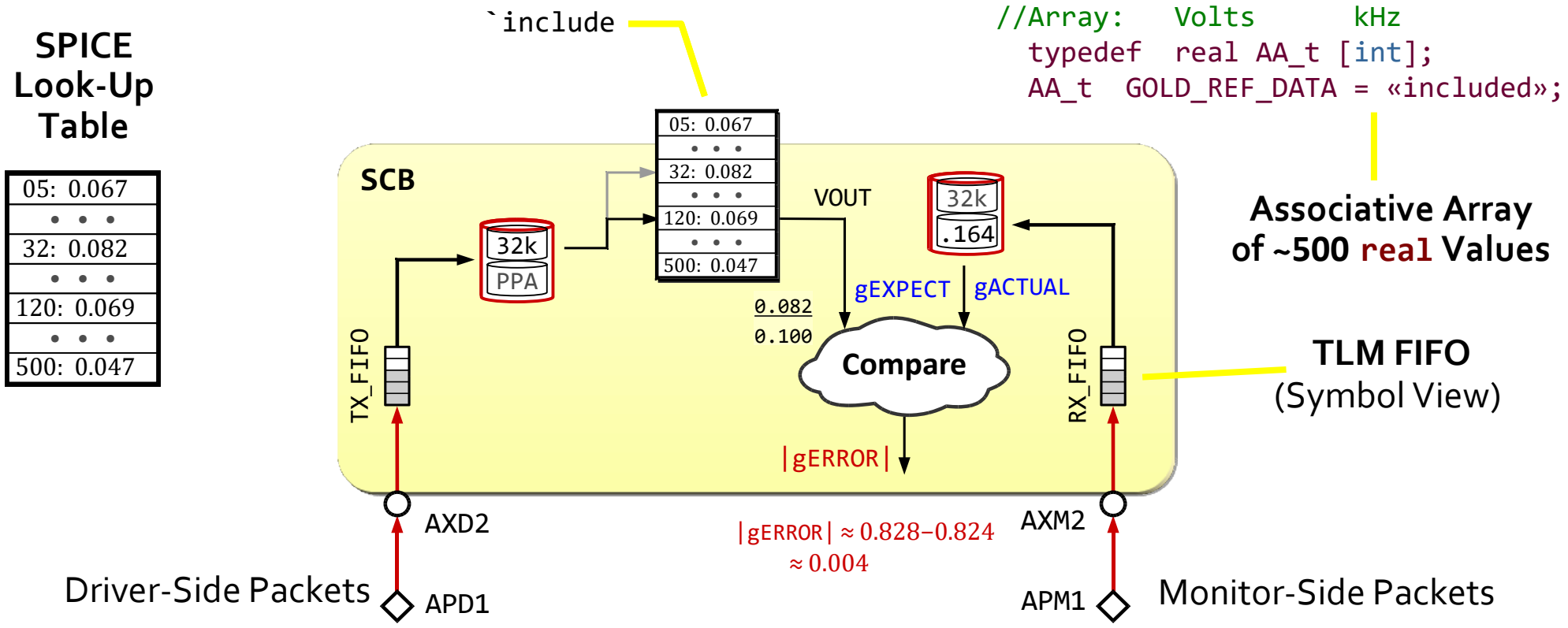
- Each agent typically handles a specific bus interface to the DUT.
- This agent creates monitor MON, and constructs TLM port APM1.
- Then connects existing MON port APM0 to the agent port APM1.

§4. A UVM Scoreboard

- Scoreboard architecture
- Scoreboard setup code
- **Lab #7**: Find Worst |gERROR|

05: 0.067
• • •
32: 0.082
• • •
120: 0.069
• • •
500: 0.047

Scoreboard Architecture



- SCB compares actual measured gain against expected.
- Looks up gEXPECT from the included SPICE reference table.

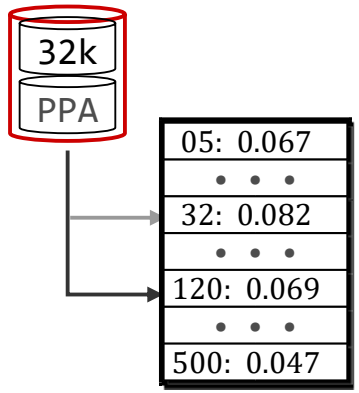
Scoreboard Setup Code

DUT-Specific Scoreboard

Set-Up Code

Task Code

```
class SCOREBOARD extends uvm_scoreboard;
  . . . . . .
  AA_t GOLD_REF_DATA;     //Look-up table.
  real PPA_IN, PPA_OUT;   //Measured amplitudes.
  PACKET TX_PKT, RX_PKT; //DRV-side, MON-side.
  «TLM exports AXD2, AXM2»
  «uvm_tlm_analysis_fifo TX_FIFO, RX_FIFO»
  . . . . . .
  «function void build_phase(), connect_phase()»
  «task run_phase()»  //Call GOLD_REF_GAIN(fINT).
  «function real GOLD_REF_GAIN(int fINT_arg)»
  «function void extract_phase()» //Max |gERROR|.
  . . . . . .
endclass: SCOREBOARD
```



GOLD_REF_GAIN(32 kHz)
→ 0.824 : **Expected Gain**

- Task run_phase() compares gEXPECT versus gACTUAL.
- Calls GOLD_REF_GAIN() to look up gEXPECT expected for fINT.
- Next packets are retrieved from FIFOs by calling .get().

Lab #7: Find Worst |gERROR|

No Return From void Function: Assign result to **gWORST**.

```
(* File = SCB_PKG.sv, Line = 194 *)

/* Find worst-case |gERROR| value across a run.
 * Errors were stored in array gERROR[1:TRIALS].
 * Return worst case to class variable gWORST.
 */
  real gWORST = 0.0;
  function void extract_phase()
    real gERROR_ABS;
    «Loop statement»
    begin:EXTRACT
      «Take absolute value of element.»
      «Compare to gWORST, then update.»
    end:  EXTRACT

  endfunction: extract_phase
```

Scoreboard Function

gERROR[1:TRIALS]

1: 0.0099
2: 0.0059
3: 0.0097
4: 0.0108
5: 0.0111
6: -0.0003
• • • • •
11: 0.1043
12: 0.0066

- Replace angle-bracketed hints with SystemVerilog code.
- Run **`make lab7`** to check your syntax

Lab #7: Solutions

```
(* File = SCB_PKG.sv, Line = 194 *)

/* Find worst-case |gERROR| value across a run.
 * Errors were stored in array gERROR[1:TRIALS].
 * Return worst case to class variable gWORST.
 */
  real gWORST = 0.0;
  function void extract_phase()
    real gERROR_ABS;
    foreach(gERROR[L])
    begin:EXTRACT
      gERROR_ABS = (gERROR[L] < 0.0)?
                   -gERROR[L] : +gERROR[L];
      if (gERROR_ABS > gWORST)
        gWORST = gERROR_ABS;
    end:  EXTRACT

  endfunction: extract_phase
```

gERROR[1:TRIALS]

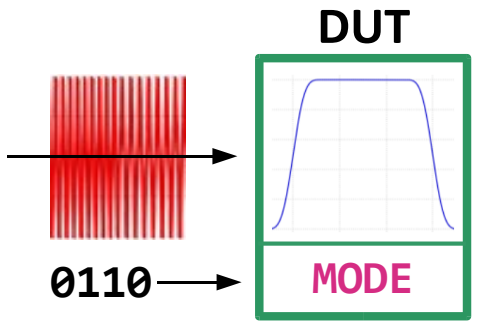
1: 0.0099
2: 0.0059
3: 0.0097
4: 0.0108
5: 0.0111
6: -0.0003
• • • • •
11: 0.1043
12: 0.0066

```
UVM_INFO @ 14 ms:  E.SCB [SCORE]
Worst-case |gERROR|:  0.01129
```

- The foreach loop is independent of the number of TRIALS.

§5. Run the Test Suite

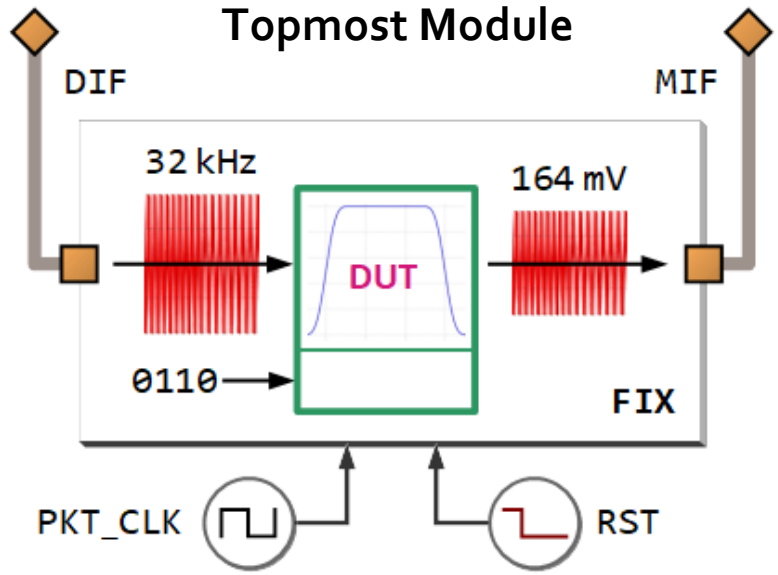
- Topmost UVM module
- A test-suite component
- UVM testbench topology
- **Lab #8:** Final testbench run



Topmost UVM Module

UVM_TB

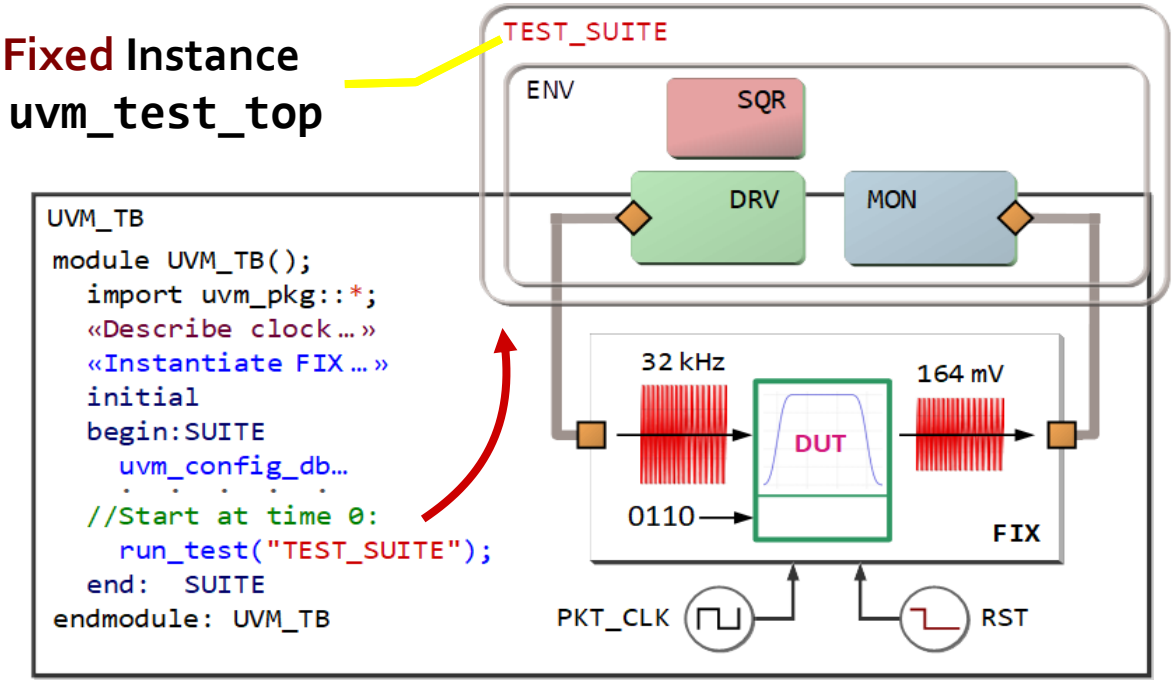
```
module UVM_TB();
  import uvm_pkg::*;
  «Describe clock, reset.»
  «Instantiate FIX, DIF, …»
  initial
  begin:SUITE
    uvm_config_db…::set(…);
    . . . . .
  //Start at time 0:
    run_test("TEST_SUITE");
  end:  SUITE
endmodule: UVM_TB
```



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

A Test-Suite Component

Gets Fixed Instance Name `uvm_test_top`



Instantiated by Factory at Time 0

Guideline:
Can alternatively specify any test-suite class name from the command line using option:
`+UVM_TESTNAME=``TEST_SUITE.`

- The component `TEST_SUITE` is never manually instantiated.
- Specify "`TEST_SUITE`" as the string argument to `run_test()`.
- Factory then creates an instance of specified class at time 0.

UVM Testbench Topology

Class-Based Hierarchy

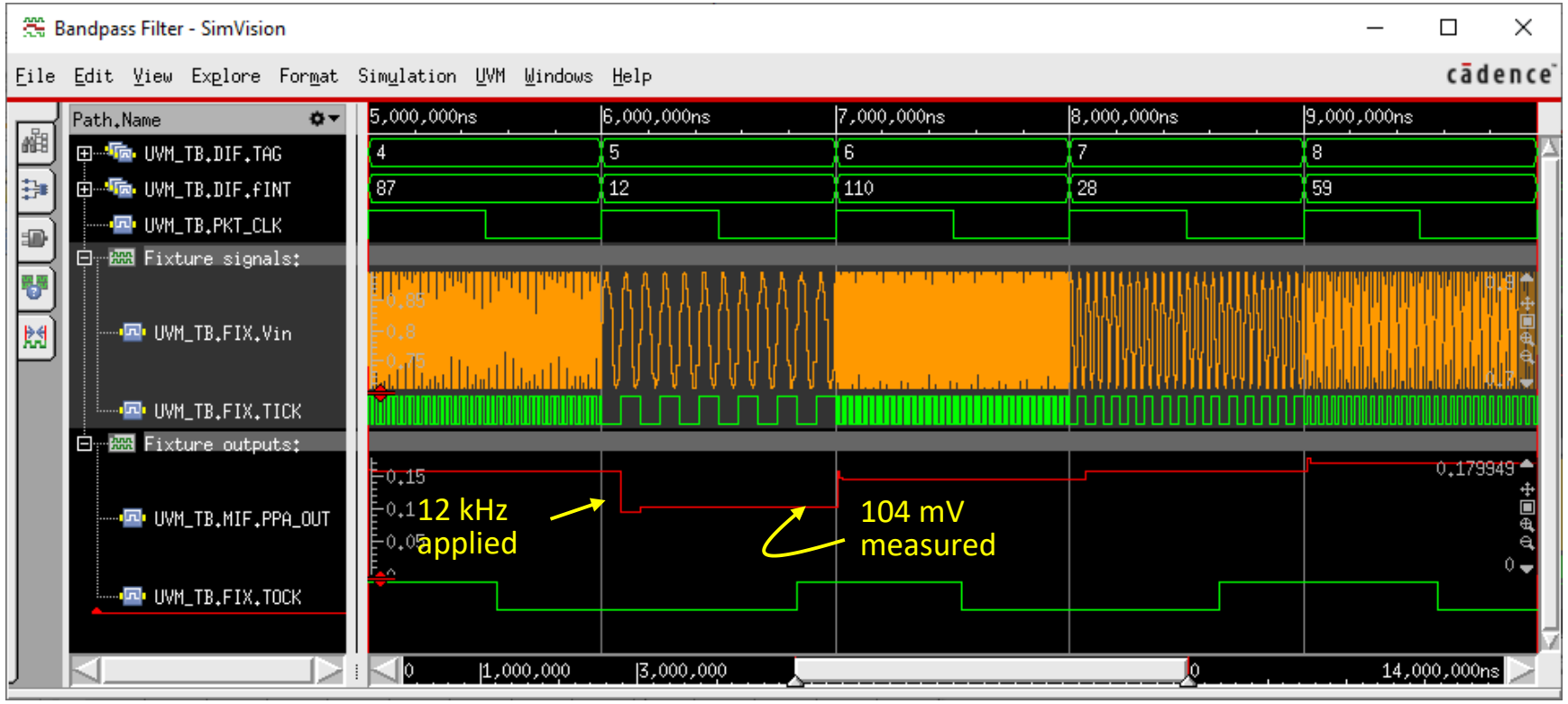
UVM Testbench: TRIALS = 10

```
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
------------------------------------------------------------
```

Not All Details Shown

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

Lab #8: Final Testbench Run



- To run simulation: `make runsim`
- To run simulation in GUI mode: `make runsim_gui`

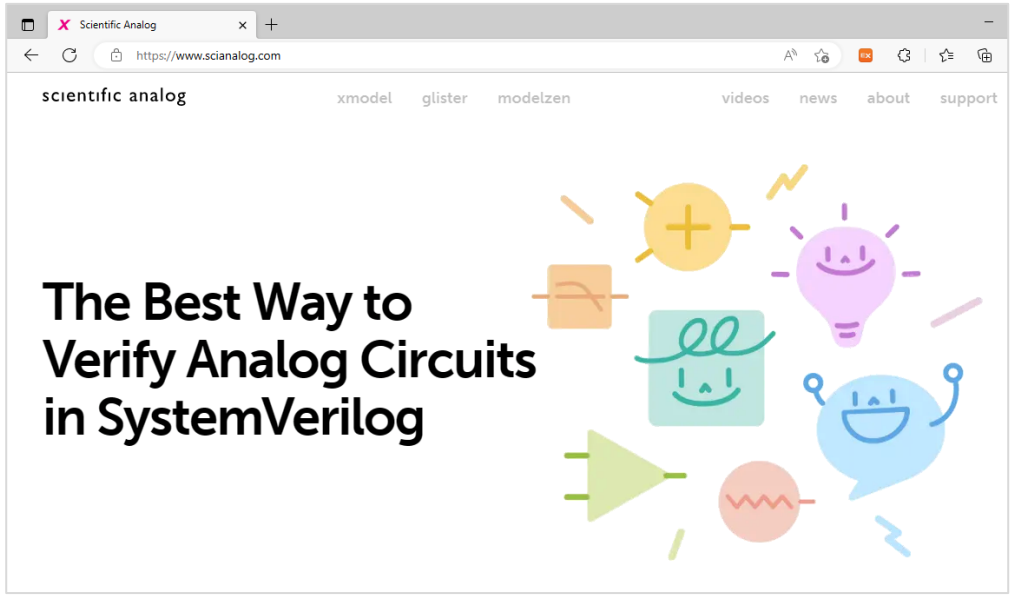
Part II: Summary

- We learned how to write UVM testbenches for AMS circuits:
  - e.g. checking the gain of a bandpass filter circuit at randomized frequencies
  - UVM testbench is built with UVM components derived from their base classes
  - The UVM components talk to each other by sending packets via TLM pathways

- This UVM testbench is no different from the ones for digital circuits!
  - Except for the fixture module enclosing the analog DUT & instrumentations
  - Composed with SV models extracted by *MODELZEN* & *XMODEL* primitives
  - The UVM simulation on AMS circuits runs efficiently entirely within SystemVerilog

To Learn More

- To learn more about *XMODEL*, *GLISTER*, & *MODELZEN*, visit Scientific Analog's website: https://www.scianalog.com



- And check out these resources:
  - Videos: scianalog.com/saflix
  - Webinars: scianalog.com/webinars
  - Newsletters: scianalog.com/avm
  - Online demos: scianalog.com/glister_demo & modelzen_demo