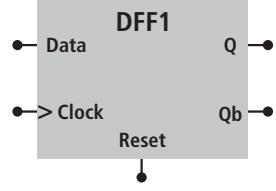
Additional Model Coding Examples

In a typical design flow it is unlikely that the same model would be developed in many different formats. The more common approach is to develop models for blocks in the format that makes the most sense for that particular type of model. In this section, a more representative model is presented for each of the four modeling approaches within the Verilog-AMS framework:

- A digital model of a D-type flip flop,
- An analog model of an operational amplifier,
- An AMS model of an analog-to-digital converter,
- A real number model of a discrete time low pass filter.

Digital: Verilog D-Type Flip Flop

The Verilog language defines logical relationships. Here is an example of a model for a D-type flip flop written in the standard Verilog format. This example defines that on the leading edge of the clock input, the data input will be read and passed to the Q



output, while the Qb output is always the complement of the Q output. The model also has an asynchronous Reset input, and defines that Q will always be zero whenever the Reset input is high.

Here is a simple description of the D-FF model:

```
module DFF1 (Q, Qb, Data, Clock, Reset);   // name of module, list of pins
 output Q,Qb;                          // normal and inverted output
 input Data,Clock,Reset;               // data in, clock, asynch reset
 reg Q;                                // register for output signal to drive
 always @(posedge Clock or posedge Reset) // when clock or reset go high
 if (Reset) Q = 1'b0;                  // if reset is high, output goes to zero
 else   Q = Data;                      // otherwise on clock edge get data
 assign Qb = ~Q;              // define notQ output to be inverse of Q
endmodule
```

The output Q is updated whenever there is a positive edge on either the Clock or the Reset input. It updates the Qb node using an assignment statement to continuously keep Qb equal to the inverse of the Q signal (Qb is not declared to be a reg because it isn't driven by behavioral code).

Here is a simple testbench to check out the operation of the flip-flop:

```
`timescale 1ns/1ps
module DFF1_TB;
reg D=1,CK=0,RST=0;
DFF1 DUT(.Q(Q), .Qb(QB), .Data(D), .Clock(CK), .Reset(RST));
always #5 CK=!CK;
initial begin
 #13 D=0;
 #15 D=1;
 #10 RST=1;
 #10 RST=0;
 #22 $stop;
end
endmodule
```

This is the top-level testbench for this simulation so no external pins are needed on module DFF1_TB. The module defines registers for the signals that will be driven, instantiates the DFF1 module as instance DUT, drives a 100MHz square waveform into the Clock input, and changes the Data and Reset inputs a few times to verify that the block operates as expected.

Note that the discrete timescale is specified for this testbench module. All discrete or mixed-signal Verilog blocks will use some timescale to define the time units and resolution for the discrete time simulation (either directly defined with the module as shown here, or in a standard include file, or defined as a compilation option). All of the previous examples in this chapter would only react when an input changed, so they would work with whatever timescale was appropriate for the rest of the system. However, in this example, the module is using the # operator to define delays within the system, and so proper operation of the module requires the time unit to be defined here. In this case the time unit is defined to be nanoseconds, so that the #5 is a 5ns delay (rather than, for example, 5ps or 5 seconds). Care should be taken to choose a timescale and resolution for the project so that the modules are consistently defined throughout. Typically this

is chosen based on standard practice in the digital models (most commonly with a time unit of 1ns or 1ps), and a resolution chosen based on project requirements (may range from 1ns to 1as depending on system operational speed). In the remaining examples in this chapter, a timescale of 1ns/1ps is assumed.

Simulation of the `DFF1_TB` module results in the waveform shown in Figure 4 below. Note that because `Q` was not initially defined, its value is unknown (X) until the first positive edge of `Clock`, where it is assigned the value of the `Data` input. When `Data` goes low or high, `Q` becomes that value on the next positive edge of `Clock`. When `Reset` goes high, the output immediately changes to low, and clocking the block while it is in the reset state does not affect the output. After the `Reset` is released, the next clock input again transfers the value of the `Data` input to the `Q` output. The `Qb` output is always the inversion of the `Q` output.

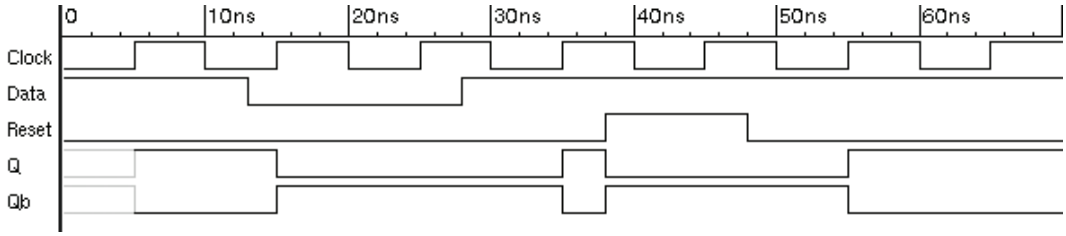
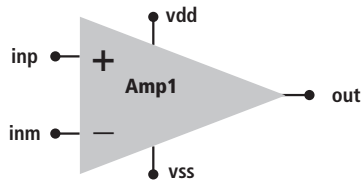


Figure 4: Simulation Results of D-Type Flip Flop

Analog: Verilog-A Operational Amplifier



Analog behavioral modeling is very closely related to circuit design. Rather than designing a system out of transistors and other standard physical circuit elements, an analog behavioral model is constructed out of general dependent sources that can have any functional relationship between their inputs and the output voltage or current. Those relationships could include linear, nonlinear, integral or derivative dependencies. The models can also define internal variables that are continuously dependent on voltages and currents (and time), or variables that change discretely when specific events occur (at specified timepoints, when a voltage passes a threshold, etc.). The outputs of the model can then be described in terms of those internal variables.

An analog behavioral model for an operational amplifier may include any number of possible characteristics. The amplifier model defined here implements the DC gain, offset, clipping, and output impedance characteristics, plus the dominant pole frequency response.

To develop a model of this type, an appropriate topology must be developed that has the desired characteristics, then the behavioral model can be written to describe that topology. In the topological description shown below, the input current source `Icur` defines a linear relationship between the input differential voltage and the drive current of the output stage. This is applied to resistor `Rres`, which generates the DC gain. When the differential input is zero, series source `Vctr` provides the DC output voltage centered midway between the power supplies. Capacitor

`Ccap` is added to provide the dominant pole frequency response. Nonlinear source `Ilim` is added to shunt excess current between node `N` and ground whenever the voltage nears the supply limits. (This functions like diode limiters, using an exponential current vs. voltage relationship that increases rapidly as the voltage on node `N` gets close to either `vdd` or `vss`). Finally the series resistance `Rout` adds additional output resistance: At DC the total output resistance would be the series combination of `Rout` and `Rres`, but node `N` is effectively shorted to ground in AC operation by `Ccap` and when saturated by the low impedance path through `Ilim` when the voltage limiter turns on, so in those two cases the output resistance would just be `Rout`.

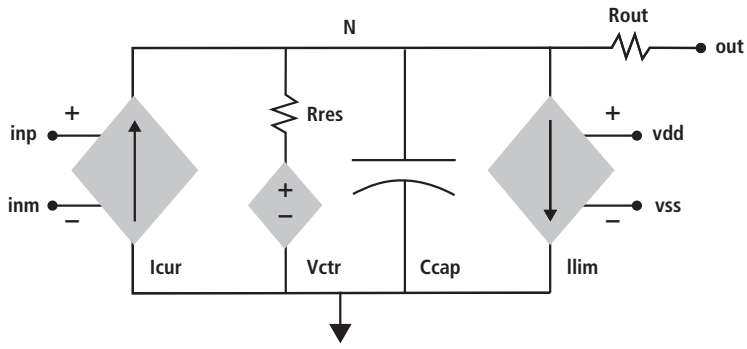


Figure 5: Topology of Amplifier Behavioral Model

```
`include "disciplines.vams"
module Amp1(inp,inm,out,vdd,vss);
 input inp,inm,vdd,vss; output out;
 electrical inp,inm,out,vdd,vss,N;
 parameter real  Gain=1k,      // DC gain (V/V)
                 Vio=0,              // input offset (V)
                 GBW=10M,            // Gain-Bandwidth product (Hz)
                 Rdc=300,            // DC output resistance (ohms)
                 Rac=100;            // AC & Sat output resistance (ohms)
 real Gm, Rres, Ccap, Vnom;          // internal variables
// Macro for diode-like exponential dependence between voltage & current:
// Returns current of isat when voltage is zero, and decreases by
// a factor of 100 for each dV decrease of input voltage:
 `define fclip(V,isat,dV) isat*exp(4.6*(V)/(dV))
 analog begin
  @(initial_step) begin              // constants computed at startup
   Rres = Rdc-Rac;                   // inner resistor value
   Gm = Gain/Rres;                         // input transconductance
   Ccap = 1 / (`M_TWO_PI*GBW/Gm);          // capacitor to get specifed GBW
  end
// Contributions of current for each branch in the topology diagram:
  I(N,vss) <+ -Gm*V(inp,inm);              // transconductance from input
  I(N,vss) <+ (V(N,vss)-V(vdd,vss)/2)/Rres;      // resistor equation: I=V/R
  I(N,vss) <+ ddt(Ccap*V(N,vss));          // capacitor equation: I=d(CV)/dT
  I(N,vss) <+ `fclip(V(N,vdd),1,40m)       // top & bottom parts of limiter
        - `fclip(V(vss,N),1,40m);
  I(N,out) <+ V(N,out) / Rac;              // series resistance on output
```

```
  end
endmodule
```

To verify the operation of the amplifier, a testbench is developed. For Verilog-A models it is fairly common to test their operation in a circuit design environment using standard elements to describe the testbench, and running several simulations to check the response under various conditions. For an amplifier, typical tests would usually include simulation with closed loop feedback and simulation in an open loop configuration, and should include both checks for linear region operation as well as saturation when the output reaches the supply levels. Typically these tests are performed in several simulation runs, each concentrating on specific aspects such as DC transfer curve, AC small signal, and transient analysis. A test schematic is shown below. In this testbench, the feedback resistor values are defined as design parameters (`R1` and `RF`), so that open and closed loop testing can be performed simply by adjusting the size of the resistances.

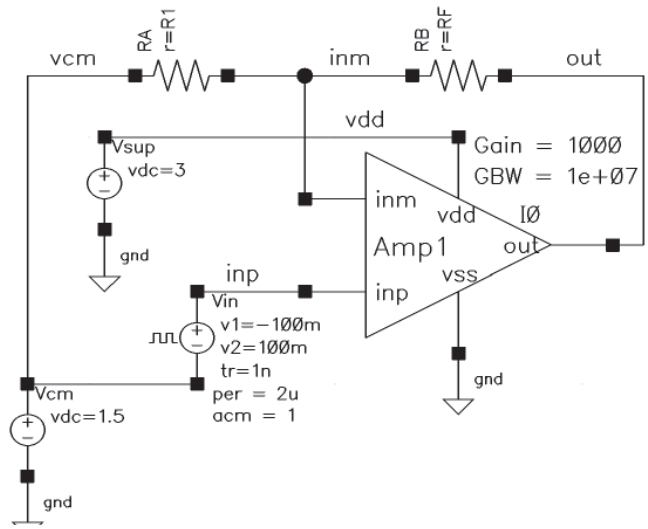


Figure 6: Schematic Testbench for Amplifier

A DC sweep simulation can be performed to verify the DC transfer characteristic. In particular, it is useful to verify continuity between the linear and saturated regions. Testing in closed loop configuration also verifies model convergence in the presence of significant feedback.

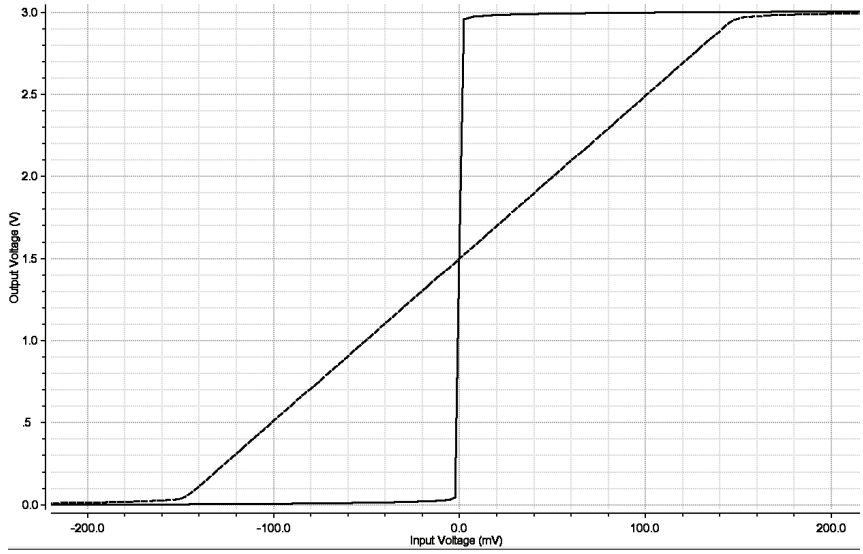


Figure 7: DC sweep – Open Loop (RF=1GΩ, R1=1Ω), and with Closed Loop Gain of 10 (RF=90KΩ, R1=10KΩ)

An AC analysis can be performed to verify frequency response. Open loop testing verifies small signal gain, corner frequency, and phase relationships. Closed loop response tests for proper gain and phase in the presence of feedback.

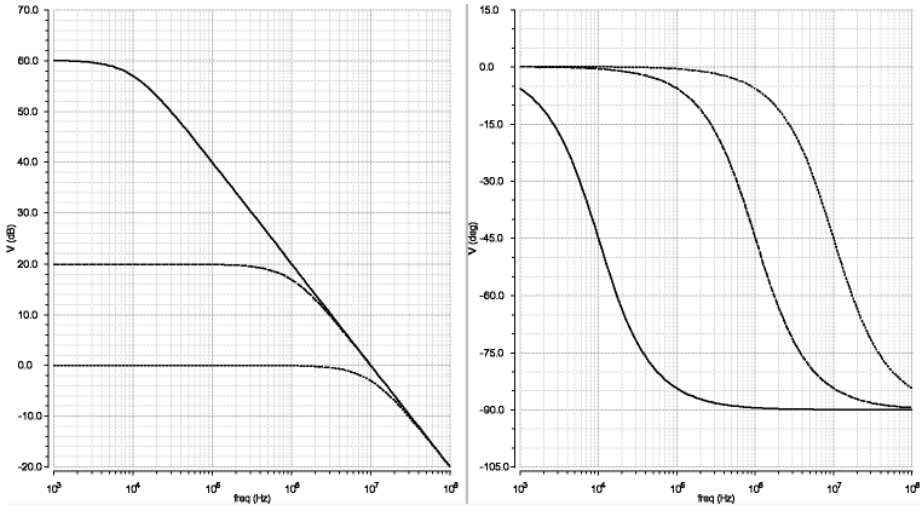


Figure 8: AC Analysis Results: Magnitude & Phase Transfer Characteristics – Open Loop, and with Closed Loop Gains of 10 and 1 (R1=1GΩ, RF=1Ω)

The transient response to a step change of input provides a direct view of the linear and nonlinear interactions. In the example below, the input is stepped between –100mV and +100mV. In unity gain configuration, the output follows the input with a pole at the gain-bandwidth product

frequency (10MHz), so the response is rapid. With a gain of ten, the output is ten times larger but responds only one tenth as fast. Open loop response slews at a limited rate (dependent on input overdrive) but saturates at the power supply limits.

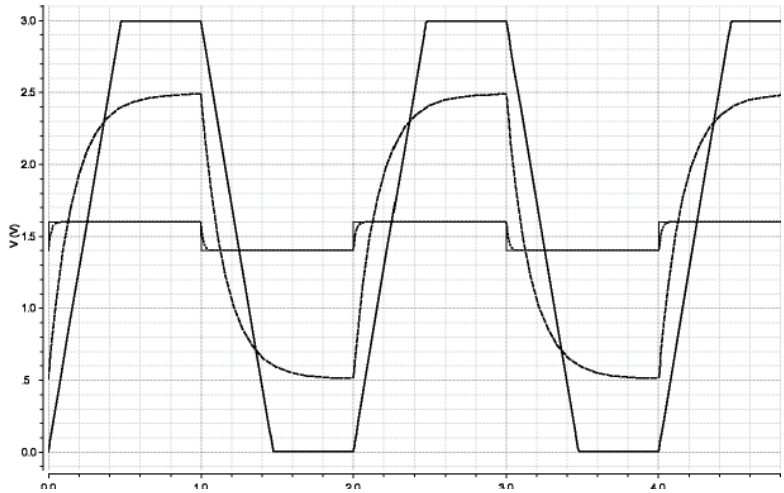


Figure 9: Transient Analysis Result for +/-100mV Step Input – Open Loop and with Closed Loop Gains of 10 and 1

While it is common to define analog testbenches in this format, it is not necessarily the best approach. A sequence of similar measurements could be coded into a Verilog-A or Verilog-AMS stimulus file which would drive the module through a sequence of tests and measure the resulting output in each case. To perform the equivalent of the AC tests, the input would be driven with several cycles of a sine wave, and the peak magnitude and phase offset of the output measured from the resulting waveform. This would require more processing, but once a task is written to perform this function, it could be reused each time it is needed. Other tests could be added, for example, forcing output current and measuring the effect on output voltage to determine the actual output resistance under various operating conditions. With such an approach, a single simulation could execute all of the desired tests and provide all measured results, instead of requiring multiple simulation runs which necessitate visual inspections of each separate output plot to estimate the response characteristics.

Mixed-signal: Verilog-AMS Digital to Analog Converter

Mixed-signal models often have a lot of digital control, in addition to significant analog functionality. Verilog-AMS is a natural language for such a model because it allows both portions of the module to be written in the format that is most natural for the tasks to be performed (as was shown in the previous PGA example). A digital to analog converter (DAC) is now described. In this model, the primary input is a digital input bus. Whenever the input changes, the output voltage is computed as a proportion of the reference supply voltage.

When defining analog outputs that are controlled by discrete signals, care needs to be taken to properly condition the signal so that it will be compatible with the analog simulation environment. If the analog output is defined to discretely step from one value to another, significant numerical problems can occur over that discrete change. This can slow down the simulation or even cause a convergence failure. For proper operation, the discretely changing quantity should be passed through a filter function to convert the step change to a ramp or other continuously time-varying waveform. This allows the analog simulator's variable-timestep algorithm to take smaller steps when needed to incrementally follow the changing waveforms. In this model, a parameter is included for rise time which will be applied to the discrete change using the built-in transition filter.

It is also important to ensure that analog signals are driven with valid values. It is not valid to drive an analog voltage or variable to an "X" state, so any measurements taken from the digital context and used to drive analog must be checked for validity. In this model, any illegal bus value will result in an output voltage of zero.

```
`include "disciplines.vams"
module DAC6 (Din,Aout, Vdd,Vss);
input [5:0] Din;              // digital input bus
output Aout;                  // analog output
input Vdd,Vss;                // reference supply for output
electrical Aout,Vdd,Vss;

parameter real tr=10n;        // (sec) risetime for output changes
parameter real rout=1k;       // (ohms) output resistance
real kout;              // output as fraction of supply
real vout;                    // continuous analog output voltage

always begin
 if (^Din === 1'bx) kout=0;   // if any bits invalid in Din, output is zero
 else kout = Din/63.0;        // else compute scale factor for output
 @(Din);                      // repeat whenever Din changes
end

analog begin
 vout = V(Vdd,Vss)*transition(kout,0,tr,tr); // ramp to fraction of supply
 I(Aout,Vss) <+ (V(Aout,Vss)-vout)/rout;         // drive output
                                                 voltage+resistance
end
endmodule
```

Testing the DAC operation is simple when the testbench is written in Verilog-AMS. Analog voltages and discrete (logical or real) signals can be both driven and measured directly. One useful approach is to define the analog stimulus to be controlled by a few discrete variables, so a single `initial` block can be used to control both the discrete and analog stimulus. This would take the form of a sequential list of variables to be set, and delays to wait for the time of the next set of changes. For this model we drive the supply voltages, cycle through all values on the digital input, and display the resulting output.

```
module DAC6_TB;
electrical Aout,Vdd,Vss;            // analog pins
real Vsup,Vgnd,Tr;            // control variables to drive analog supplies
reg [5:0] Din;                // digital input bus
```

```
DAC6 DUT(Din,Aout, Vdd,Vss); // instanciate the digital to analog converter

analog begin                          // analog drive description:
 V(Vdd) <+ transition(Vsup,0,Tr);     // drive the analog supplies
 V(Vss) <+ transition(Vgnd,0,Tr);     // using the control variables
end

initial begin                 // This is the sequential test procedure
 Din=0;                       // Initialize digital input bus to zero
 Vsup=3; Vgnd=0;              // Initial 3 volt supply and ground voltage
 Tr=200n;                     // risetime for supply changes
 repeat(63) #50 Din=Din+1;    // step Din thru all codes, incrementing every 50 ns
 #100 Din=8'b100000;          // step to half scale input
 #100 Din=8'b10xx00;          // check response for X input
 #100 Din=8'b100000;          // back to half scale
 #100 Vsup=2.4;               // Check that output follows supply midpoint
 #300 Vsup=3.0;
 #300 Vgnd=0.4;               // Check that it follows negative reference too
 #300 Vgnd=0;
 #300 $stop;                  // done with testing
end
endmodule
```

Note that a few lines were added to the end of this test which check the output when the input becomes invalid and when the power supply voltages change. These are standard checks that should be included whenever verifying behavioral models at the block level because it is easy to forget that atypical inputs can occur and inadvertently make assumptions in the model that aren't universally accurate. A few lines of code can check that the model functions appropriately when given typical as well as unexpected inputs.

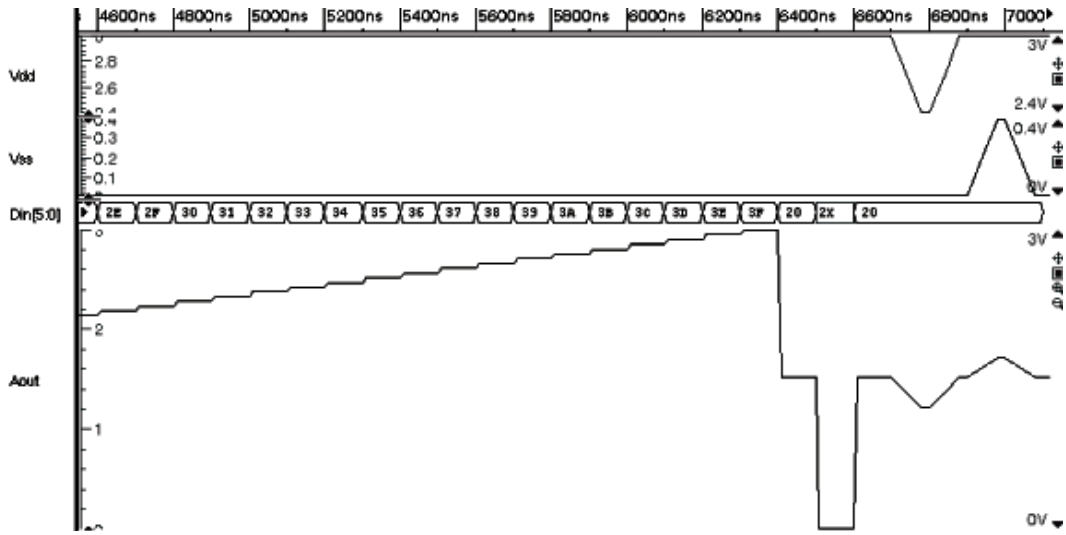


Figure 10: Portion of Transient Results from Digital-to-Analog Converter Simulation

In a more sophisticated model, specific requirements for bias inputs, nonlinearities and/or random offsets on the digital to analog conversion, or operational specifications that require measurement of specific distortion measures in the output signal might be included. This would require a more comprehensive testbench that would generate a particular input signal, sample the resulting output waveform, and perform the desired calculations. Using the Verilog-AMS language, this is simply a coding exercise, as all of the drive and measurement capabilities are very naturally describable in the language.

A recommended approach to testing is to define tasks that perform particular tests and report whether the output is within specification. For this example, a task could be defined to place a specific value on the digital inputs and check if the output value is correct:

1. Drive `Din` to a specified value,
2. Wait a specified time delay for the output to respond,
3. Compute the expected output,
4. Compare actual output to expected output and print message if it falls outside of a specified tolerance,
5. Count the number of tests that pass or fail.

Here is a task that can perform this operation, checking that each value is within 1% of its expected value:

```
integer CountErr=0,CountOK=0,i;     // variables to keep running count of tests
task D2Acheck;                      // task to check for valid response
 input real Dval;
 real Anom,PctErr;
 begin
  Din = Dval;
  #50 Anom = V(Vdd,Vss)*Din/63;
  PctErr = (V(Aout,Vss)-Anom)/V(Vdd,Vss)*100;
  if (abs(PctErr)>1.0) begin
   $display(
   "*SpecErr: Din=%b Anom=%5.3f Aout=%5.3f Out of spec by %.1f%% at T=%.1fns",
          Din,  Anom,   V(Aout,Vss),    PctErr, $realtime);
   CountErr=CountErr+1;
  end
  else CountOK=CountOK+1;
 end
endtask
```

Using such a task, the test procedure could be written to check specific codes to verify proper operation.

```
initial begin                       // This is the sequential test procedure
 Vsup=3; Vgnd=0;                    // Initial 3 volt supply and ground voltage
 Tr=100n;                           // risetime for supply changes
 for (i=0; i<64; i=i+1) D2Acheck(i);       // check all 64 codes
 D2Acheck(32);                      // check midrange point
 Vsup=2.4;      #100                      // ramp to new supply voltage
 D2Acheck(32);                      // recheck midrange point
 Vsup=3; #100
```

```
  Vgnd=0.4; #100                    // switch to different negative reference
  D2Acheck(32);                     // recheck midrange point
  #100 Vgnd=0;
  #100 if (CountErr==0) $display("*SpecPASS - DAC8: all %d tests passed",CountOK);
    else $display("*SpecFAIL - DAC8: %d failures out of %d tests",
                 CountErr, CountErr+CountOK);
  #1 $stop;                         // done with testing
end
```

This type of test is particularly useful when the test needs to be run repeatedly due to specification changes. The testbench reports whether the model being tested matches a defined specification rather than requiring manual visual inspection to verify that the response "looks right" after each simulation. The same testbench could also be used to test the transistor-level view of the cell, and to verify that each view matches the specification within defined tolerances. While it does take more time to develop code for such verification procedures, this approach can help to significantly automate the verification process. Once such procedures are put into standard practice in the analog and mixed-signal verification processes, they can be readily promoted for use with system-level verification procedures.

Real Number Modeling

In Real Number Modeling (RNM), analog voltages are represented as a time-varying sequence of real values. This is actually very similar to what analog simulators do. The difference is that in a typical analog simulator, the models define a set of equations. The simulator augments this via addition of topology constraints using Kirchoff's laws, and then it solves the overall constrained system(s) of simultaneous equations at each timestep to compute the voltages and currents from those equations. In a discrete real environment, there is no voltage vs. current equations, there are no Kirchoff's laws, and there is no simultaneous equation solution step – the output is *directly* computed from the input, ignoring currents and other feedback mechanisms that could have caused interdependencies between drive and load in an electrical environment.

This may appear to be too abstract. It is certainly a big step away from transistor-level simulation. However, when defining behavior at a level significantly higher than the transistor-level, much of the system is described in terms of direct relationships: when something occurs at an input, it is processed by the block (compared, scaled, translated, filtered, delayed, slewed, etc.) to produce an output signal, which is then processed by the block it is connected to. Thus the concept of discrete RNM is highly applicable to system investigation through system verification – which constitutes most of the verification testing that is needed. It is already common practice to verify subsystems at the transistor level, and then use behavioral models in higher-level simulations, so it is a natural extension to create that behavioral model using real rather than AMS modeling techniques to provide optimal throughput in all of the remaining higher-level simulation tasks.

The concept of RNM is straightforward. If the input/output relationship is a direct transfer characteristic, a mathematical expression can be written that describes how to update the output whenever the input changes. Checking for proper biasing is also simple. The power supply, bias current and voltage inputs would be passed into the model as real numbers. The simulator

would check to see if they are within reasonable tolerance, and the outputs would only be driven if the proper bias and control are applied. These operations were previously described in the programmable gain amplifier example.

Time-domain response waveforms do require more work to define in RNM. While the standard analog modeling languages have built-in functions for operations such as rise time insertion, slewing, integration, differentiation, and analog filtering, those operations are not automatically available when working with real number signals. Instead, a per-timestep format for implementation in the discrete real domain is required. Many of these effects can be written in a few lines of code, but some experience is required. When getting started with RNM, build up a library of these types of functions for use as building blocks, and this will help to develop the experience and to understand how to use them effectively. Once that experience has been acquired most transfer characteristics, which were described in analog terms, can readily be converted to an equivalent discrete real implementation.

An integration operation is often used in analog systems. In the discrete domain, given an input that is a sequence of timepoints, the output can be computed based on the previous and current values of the input using the standard trapezoidal integration formulation:

$$OUT_{NEW} = OUT_{OLD} + (T_{NEW}-T_{OLD})\ (IN_{NEW}+IN_{OLD})/2$$

So as long as the input is changing, this allows direct calculation of the output (OUT_{NEW}) at each new timepoint (T_{NEW}, IN_{NEW}).

Verilog-AMS/wreal Low-pass Filter

One particularly useful example of this type of discrete-real formulation of a time-domain response is the definition of a low-pass filter. Filters, in the analog domain, are usually defined in the frequency domain as a rational polynomial `H(s) = N(s)/D(s)` that has poles and zeros. This approach works well in the frequency domain, and can be mapped to RLC circuits very usefully. But in a strict event-driven real modeling sense, that form is fairly abstract. In discrete systems, the typical approach to filtering is in the Z domain, which is used in sampled-data system analysis. The filter defines the output in terms of the input and unit-delayed versions of the input and output.

Conversion between `H(s)` and `H(z)` is a standard technique [9]. The primary problems associated with implementing filtering in a real model are in getting the input into a sampled format and applying the Z-domain transfer function properly to the signal.

For example, consider a first order low-pass filter's standard s-domain transfer function:

```
H(s) = OUT / IN = 1 / ( 1 + s/Wp)
```

where `Wp` is the corner frequency in radians per second. The standard textbook conversion format from `H(s)` to `H(z)` uses the Bilinear Transform [10]:

$$s = (2/Ts)(1- z^{-1})/(1+ z^{-1})$$

where `Ts` is the sampling rate and z^{-1} is the unit delay operator.

Applying this to the above equation results in:

$$H(z) = 1 / (1 + (2/WpTs)(1- z^{-1})/(1+ z^{-1}))$$

Multiplying through by $(1+z^{-1})$ and collecting terms results in:

$$H(z) = (1 + z^{-1}) / (1 + 2/WpTs + (1- 2/WpTs)z^{-1}) = (n_0 + n_1z^{-1}) / (d_0 + d_1z^{-1})$$

where,

$$n_0 = 1,\ n_1 = 1,\ d_0 = 1 + 2/WpTs \text{ and } d_1 = 1 - 2/WpTs$$

The next step is to convert the general Z-domain transfer function into code that will compute the result using a difference equation:

$$H(z) = OUT / IN = (n_0 + n_1z^{-1}) / (d_0 + d_1z^{-1})$$

Cross multiplying can be used to solve for the relationship between IN and OUT:

$$OUT \bullet (d_0 + d_1z^{-1}) = IN \bullet (n_0 + n_1z^{-1})$$

Next, the OUT value can be solved for in terms of the input and previous values (remember, z^{-1} is the unit delay operator):

$$OUT = (n_0IN + n_1z^{-1}IN - d_1z^{-1}OUT) / d_0$$

This can then be converted to the desired format by replacing the z^{-1} multiplier by the previous value of its argument:

$$OUT_{new} = (IN_{new}*n_0 + IN_{old}*n_1 - OUT_{old}*d_1) / d_0$$

A Verilog-AMS/wreal behavioral model of a low-pass filter using this transfer characteristic is shown below.

```
`include "constants.vams"
`timescale 1ns/1ps
module LPF1s(OUT,IN);
output OUT; input IN;
wreal OUT,IN;
parameter real Fp = 10M;            // corner frequency (Hz)
parameter real Ts = 10n;            // sample rate (sec)
real d0,d1,ts_ns;
initial begin
  ts_ns = Ts/1n;             // sample rate converted to nanoseconds
  d0 = 1+2/(`M_TWO_PI*Fp*Ts);             // denominator coefficients
  d1 = 1-2/(`M_TWO_PI*Fp*Ts);
end
real INold,OUTval;                  // saved values of input & output
always #(ts_ns) begin               // sample input every Ts seconds
  OUTval = (IN + INold - OUTval*d1)/d0; // compute output from input & state
  INold = IN;                       // save old value for use next step
end
assign OUT = OUTval;                // pass value to output pin
endmodule
```

The response to a step change of input can verify that the output has the proper time response. The simple stimulus module shown below drives a 1-volt square wave into three instances of the low-pass filter, each with a different corner frequency. The results show the appropriate time constant response for each of the models.

```
module LPF1s_TB;       // testbench to check response of low-pass filters
  wreal OUT1,OUT2,OUT3,IN;
  LPF1s #(.Fp(10M), .Ts(1n)) LP1(OUT1,IN); // Tau = 1/2πFp = 16ns
  LPF1s #(.Fp(30M), .Ts(1n)) LP2(OUT2,IN); // Tau = 5.3ns
  LPF1s #(.Fp(100M),.Ts(1n)) LP3(OUT3,IN); // Tau = 1.6ns
  real Vin=0;                          // Initialize input to 0
  always #20 Vin=1-Vin;                      // alternate between 0 and 1 every 20ns
  assign IN=Vin;                       // drive input pin with Vin value
  initial #120 $stop;                  // stop after several cycles
endmodule
```

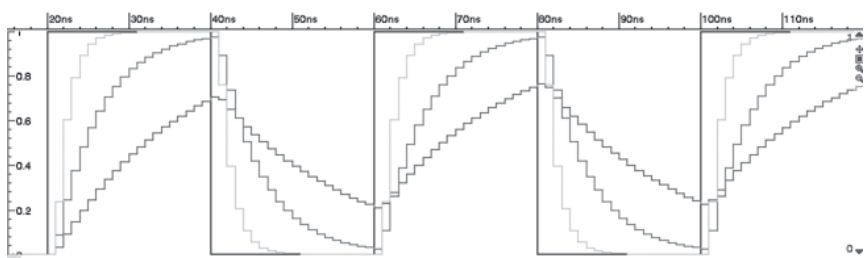


Figure 11: Response of Three Discrete Real Low-pass Filters to Square Wave Input Signal