cadence®

Verilog-AMS Real Valued Modeling Guide

Product Version 19.09
September 2019

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

1

Introduction

Virtually all modern System on Chip (SoC) designs of today are mixed-signal designs. Mixed-signal design is one of the biggest challenges of the modern sub-micron SoC design world. Most systems have to interface their millions of gates, DSPs, memories, and processors to the real world through a display, an antenna, a sensor, or a cable. This means that they have analog and digital content. These mixed-signal SoCs have to be designed with an unprecedented level of integration, in processes that have compromised every facet of performance for size, and in a scale that exposes the designer to the ravages of physics of small dimensions.

Until recently, mixed-signal designs could be decomposed into separate analog and digital functions. Today, mixed-signal designs have multiple feedback loops through the analog and digital domains. It is not practical to decompose them into separate functions without losing essential system behavior. This requires an integrated mixed-signal simulation and verification environment. Performance and reliability are now the key problems. Verification has become the major bottleneck in the design flow.

The old way of using the tools will not work any more. It will take a lot of planning to figure out a design and, even more important, a verification strategy before starting the effort. The tools are much more capable than before, but the gap in solving analog types of problems in a pure digital environment and vice versa is greater than ever. This requires a change in people's mentality, group structure, management, working style and obviously the tools chain, working environment, and verification methodology.

Cadence has introduced a digital-centric mixed-signal verification environment – Digital Mixed-signal (DMS). This new verification environment targets customers using digital-centric use models. It refers to, but is not limited to, mixed-signal verification using only digital simulators. In other words, it delivers capabilities to verify the mixed-signal design using digital-centric methodologies. This book provides the overall benefits of the Digital-centric Mixed-signal Verification methodology and explains how this methodology can enable customers to perform top-level verification at digital speed.

The Verification Problem chapter introduces the verification problem and highlights how real value modeling can help overcome the current limitations. Appendix A: Advanced Digital Verification Methodology and Appendix B: Analog Simulation Today provide additional background information on the proposed methodology.

The chapters Verilog-AMS Wreal Features and Modeling with Wreal focus on Verilog-AMS wreal modeling. The basic language feature and advanced capabilities are described in the Verilog-AMS Wreal Features chapter, while the Modeling with Wreal chapter presents application examples of wreal models. It is recommended that you read the Verilog-AMS Wreal Features chapter first, from beginning to end; however, the sections in the Modeling with Wreal chapter are relatively independent and can be read in any order.

Note: Cadence provides various types of material for hands-on experience. The examples shown in this book and many others are included in the standard IUS/IES installation (<IUS/IES path>/tools/amsd/wrealSamples). Some additional tutorials on wreal features can be found in <IUS/IES path>/amsd/samples/aium/. There is an AMS designer workshop, a DMS workshop, and a DMS Techtorial available on request. Finally, training classes on AMS Designer and Verilog-AMS language are available as well.

2

Verification Problem

A typical SoC Verification flow involves top-level simulation of components at various levels of abstraction. The verification engineer needs to integrate modules from the analog and digital design team as well as from third-party IP providers. Each module in itself could be a mixed-signal design that is created using either an analog-centric or a digital-centric use-model. The design descriptions could be schematics, SystemVerilog, and Verilog (or VHDL)-AMS in a single top-level SoC verification. Figure 2-1 illustrates this scenario (the dark gray blocks are analog or mixed-signal blocks):

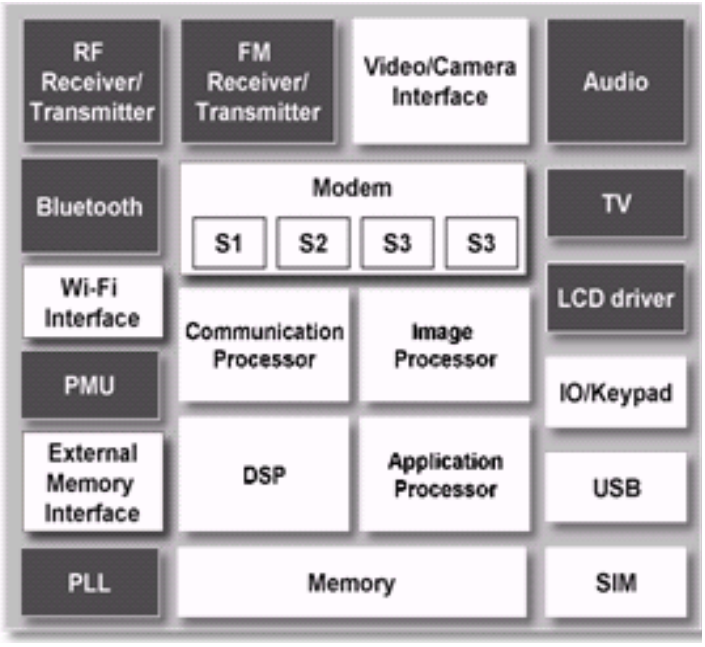


Figure 2-1 : Mixed-signal system on chip

At present, functional complexity in terms of modes of operation, extensive digital calibration, and architectural algorithms overwhelms the traditional verification methodologies. Simulation at this top-level is extremely costly (both in terms of time and licensing cost) because a significant amount of the SoC has to be simulated inside the analog engine. Finding a way to reduce the time and expense to verify the SoC, while trading off some

accuracy that is not needed at this high level of integration, is extremely important. Infact, the basic models of the analog parts might be sufficient for the verification steps.

The verification problem is hierarchical, as the design is created in a hierarchical fashion. The top-level SoC is the highest level of integration and can be tested using the top-level verification step. Verification steps are much cheaper and easier on smaller blocks because the simulation performance is better and the verification tasks are more limited. Nevertheless, the integration of multiple blocks needs to be verified to ensure that the interfaces work correctly and the blocks work together smoothly.

Generally, the verification goal is split into two parts: fulfilling the specification requirements and verifying that the system works correctly.

The main goal is always to fulfill the requirements given in the specification. This determines that the system does what it needs to do.

However, it is also important to verify what the system should not do. This includes reactions on conditions that are outside of the specification, deadlocks, power supply current peaks, oscillations, and so on. In most cases, these conditions are harder to verify because they are not explicitly stated.

The verification tasks require a certain amount of simulation data, data accuracy, and the right simulation context. For example, a detailed analysis of an RF low noise amplifier requires very high simulation accuracy, whereas checking the pin connectivity for the same block has an extremely low sensitivity towards accuracy. On the other hand, the pin connection check requires the block to be simulated in the context of the surrounding circuitry whereas the RF checks might be relatively independent of the integration.

Consequently, a full chip simulation using the highest level of simulation accuracy would be desirable to fulfill all verification requirements at the same time. However, the limiting factor in this context is simulation performance. Even though the simulation performance of current simulation tools has greatly improved (by using the fast SPICE methods), the increased complexity of designs will continue to limit detailed verification by time and simulation performance. On the other hand, very few verification goals require a high level of simulation accuracy within a full chip simulation context.

The practical way around this problem is a hierarchical verification approach that uses different levels of design abstractions for different verification goals. For the analog circuit part, common levels of abstraction are:

- Extracted transistor netlist including parasitic elements
- Transistor-level SPICE simulation
- Transistor-level fast SPICE simulation

- Analog/conservative behavioral modeling
- Real value modeling
- Pure digital model

The gain in simulation performance and the reduction in accuracy are highly dependent on the application. The real value modeling approach will be discussed in detail below. It should be noted that the Virtuoso AMS Designer does support a mixed-signal simulation including all levels of abstractions above. It also supports all common design and modeling languages.

Real Value Modeling

The target application of real valued modeling (RVM, also called real number modeling) is to bridge the gap between the performance requirements for a full chip simulation and the accuracy limitations of a mixed-signal design. A significant speed-up in simulation performance and reduction in the license cost can be achieved by replacing the analog portions of the SoC with functionally equivalent real value models that do not require the analog engine. In addition, typical analog simulation problems, such as convergence issues, very small time steps, and capacity and performance limitations are eliminated while maintaining the fundamental analog, meaning continuous value, and behavior of the circuitry. Figure 2-2 shows a modified version of Figure 2-1 with the analog portions of the design replaced with functionally equivalent real value models (shown in red):

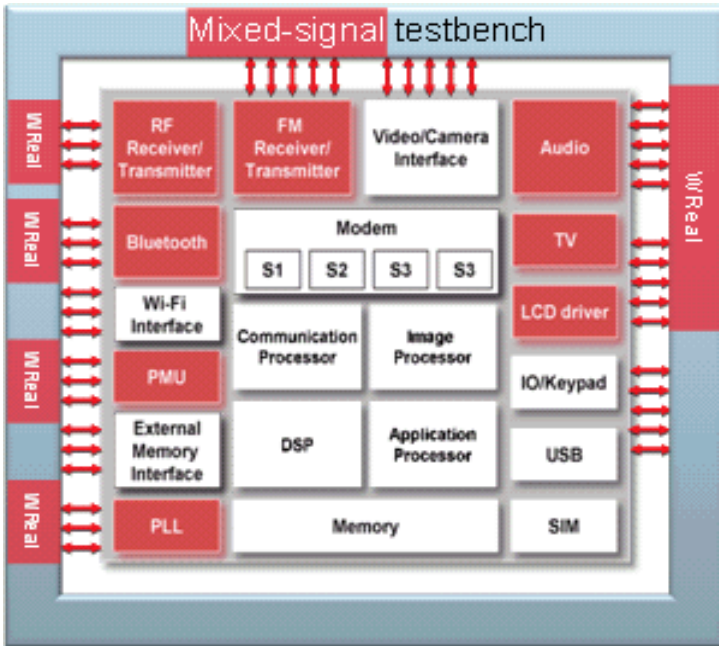


Figure 2-2 Mixed-signal SoC with wreal models and testbench

It is obvious that this top-level verification strategy is not a replacement for detailed block or multiple-block, cluster-level verification with full analog simulation accuracy. Even for the top-level verification, there might be rare cases where the RVM approach does not provide enough accuracy for a particular verification goal, such as cross talk of a global net into an analog sub block. The model verification task that compares the model against the transistor-level reference for each RVM model used in the top-level verification is essential to qualify the overall verification result.

RVM also opens the door to use other logic verification methodologies (see Appendix A: Advanced Digital Verification Methodology), such as

- Metric-driven verification (randomization, coverage, and assertion-based)
- Higher-level verification languages, such as SystemVerilog and *e*

These verification techniques are not yet fully supported for electrical systems and they require a high simulation performance to check sufficient number of data cycles. RVM can address both these issues.

RVM is a mixed approach, borrowing concepts from analog and digital simulation domain. The values are continuous, floating-point (real) numbers, as in the analog world. However, time is discrete, implying that the real signals change values based on discrete events. In this approach, we apply the signal flow concept so that the digital engine can solve the RVM system without support of the analog solver. This guarantees a high simulation performance in the range of a normal digital simulation and orders of magnitudes higher than the analog simulation speed.

There are four different language standards that support RVM. These are:

- **wreal** ports in Verilog-AMS
- **wreal** in VHDL
- **wreal** variables in SystemVerilog
- **wreal** types in *e*

It is important to note that the real-wire (wreal) is defined only in the Verilog-AMS LRM. Thus, a wreal can be used only in the Verilog-AMS block. However, it is only the digital kernel that solves the wreal system. There are no performance drawbacks when using these types of Verilog-AMS modules in a digital simulation context as compared to other real value modeling approaches, such as SV-real or VDHL-real.

Note: Real value models in *e* are mainly targeting for testbench development.

Simulation Performance, Accuracy, and Modeling Effort

For the analog subsystem, common levels of abstraction are:

- Extracted transistor netlist including parasitic elements
  - Post layout representation
  - Highest level of accuracy that can be achieved in circuit-level simulation
  - Huge amount of devices
  - Low simulation performance
- Transistor-level representation, SPICE-level simulation
  - Nominal reference for analog simulation
  - Often used as “golden” simulation reference
- Transistor-level representation, fast SPICE simulation
  - Trade off between accuracy and speed
  - Mostly close to SPICE-level simulation accuracy
  - Simulation speedup compared to classical SPICE in the range of 20x
- Analog behavioral modeling
  - Conservative behavioral models using voltages and currents
  - Written in Verilog-A, Verilog-AMS, or VHDL-AMS
  - Typical speedup compared to classical SPICE in the range of 5-100x
- Real value modeling
  - Continuous value, discrete time (event-based) models
  - Signal flow models
  - Written in Verilog-AMS, VHDL, SystemVerilog, *e*
  - Maintaining analog-like behavior
  - Can be solved in digital simulation kernel
  - Simulation performance close to digital speed
  - Compared to classical SPICE approach, a performance improvement of 100000-1000000x can be achieved.

- Pure digital model
  - Digital model for analog blocks
  - Written in Verilog, SystemVerilog, and VHDL
  - Analog signal represented as single logic value: very poor representation of analog behavior but potentially sufficient for connectivity type of checks
  - Analog signal represented as logic bus: similar accuracy as RVM, however, no match between digital and analog implementation (bus vs. scalar wire)

The gain in simulation performance and reduction in accuracy are highly dependent on the application. There is no general recommendation on what level of abstraction might be useful. This heavily depends on the verification goals. All have particular advantages and disadvantages, and thus a useful application area. For example, a very rough digital model might be sufficient for just connectivity or timing checks.

Figure 2-3 shows a general trend in the accuracy/performance tradeoff. The numbers are generic and can vary significantly for different applications.
.

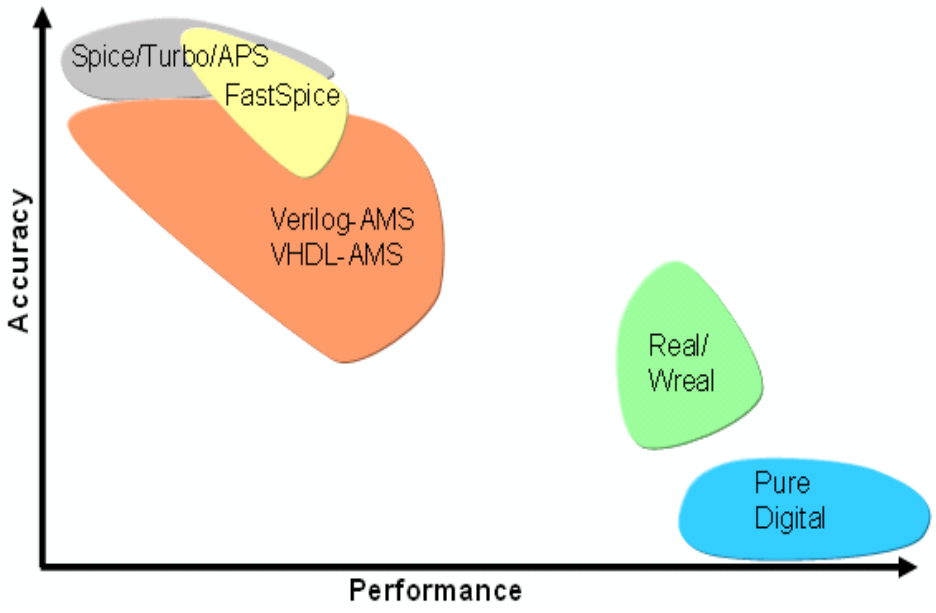


Figure 2-3 Model accuracy versus performance gain

Classical SPICE-level simulations are used as golden reference simulation. Fast SPICE engines reach the same accuracy but can also trade-off some accuracy for speed (mainly fast SPICE). Analog behavioral modeling provides a large range of capabilities, reaching from

high accuracy to high performance. However, it should be noted that a low-performance, low-accuracy model is a potential risk for inexperienced modelers. In particular, convergence issues caused by over-idealized models might slow down the overall simulation significantly. This may result in a performance decrease rather than the expected simulation speed improvement.

As mentioned earlier, RVM provides high simulation performance but restricts the model accuracy at the same time. Finally, pure digital model can be very inaccurate but might be sufficient for verification tasks including connectivity checks.

The other effect to consider in this context is the effort required to set up a simulation or create the model. Figure 2-4 illustrates the general trends.

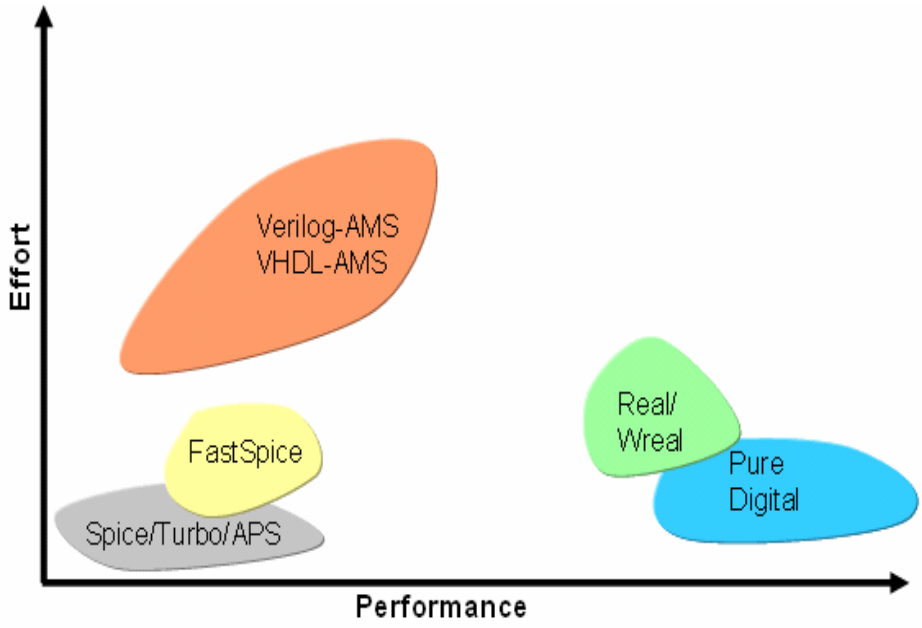


Figure 2-4 Modeling effort versus performance gain

As SPICE simulations are reference simulations, these are always executed and the effort to run this simulation must always be taken into account. Turbo SPICE and APS setup requires only one additional option (`-turbo`); so no additional work is required. Since FastSpice simulations require a detailed control on speed vs. accuracy, on a block-level basis, some amount of setup effort and understanding of the design is required.

Analog behavioral model creation effort can range from hours to days and possibly weeks to become a good behavioral model. RVM is inherently restricted to the signal flow approach and analog convergence is not an issue. Consequently, the modeling effort is significantly lower as compared to analog behavioral models. Secondly, and more importantly, the design

engineer does not need to learn a new language to create the real value models. Therefore, the ramp-up time to create these models is much lower than creating analog behavioral models. On the other hand, pure digital models – not using RVM – do have significant drawbacks in terms of functionality that limit the use for analog models (See section Why wreal? on page 24).

Behavioral models can also be differentiated by the modeling goal. A performance-oriented model needs to precisely capture critical behavior, which is necessary to efficiently explore the design space and make implementation trade-offs. Functional models capture the actual circuit behavior only to the detail needed to verify the correct design functionally. Both types of models are found in top-down and bottom-up modeling. However, the top-level functional verification is far more common in bottom-up modeling where it is used to perform the final design verification prior to tape-out. For example, the functional verification model can be used to study dynamic closed-loop functionality between the RF and digital baseband ICs, whereas a performance-oriented model of an RF block can be used for cascaded measurements, such as IP3 and explore system-level metrics, such as bit error rate and error vector magnitude.

Limitations of the RVM Approach

Real valued models are obviously limited due to the signal flow approach. On the other hand, this modeling limitation is an enabling factor that is able to solve the equations inside the digital kernel. Simple analog primitives, like a resistor, are impossible to model as real value model. The branch current through the terminals and the voltage across both pins are defined as a fixed ratio, however, there is no signal flow representation for this.

The same limitation applies for an analog RC filter. However, in most cases it is relatively straightforward to convert the analog filter characteristic, using the bilinear transform, into a discrete domain filter. A z-domain filter is easy to implement in real values. (See section Low Pass Filter on page 74).

This means, a real value model is always an abstraction above and beyond the analog behavioral details. Given the target application described above for functional verification of the analog and digital integration check, this would be a mandatory step anyway.

Detailed analog behavior, such as impedance matching, transistor sizing, continuous time feedback, low-level RC coupling effects, sensible nonlinear input/output impedance interactions, noise level, and similar effects need to be verified on the lower analog block level using conservative behavioral models or transistor-level representations.

Model Verification

Top-level verification based on real value models is only as good as the models themselves. Thus, a model verification step is mandatory for bottom-up models. In most cases, the transistor-level implementation is used as reference implementation for the real-value models. Simulation setup and testbenches are available in the analog block-level flow for these blocks.

To verify the model versus the reference, identical simulations are performed using the reference, and the model and the simulation results are compared. This can be done manually or using a newly developed tool, DCM Model Validation (DMV), to set up the simulation runs and result comparison.

A RVM-based Workflow

From a high-level point of view, a mixed-signal flow including real value modeling could look like as follows.

Concept engineering is developing the system-level view of the chip. Architectural trade-offs are made based on the system-level simulation. This is also the starting point for the baseline specification for the building blocks. For the system-level simulation, top-down models are used (See Figure 2-5). Using the real value models is an attractive possibility because it enables the connect engineer to use the same simulation environment as the detailed block implementation. Thus, the real value models are used as initial specification for the block-level implementation and are exchanged seamlessly between the concept and block-level flows.

Beside the specification and interface definitions, a high-level verification plan is developed based on the implemented features and blocks.

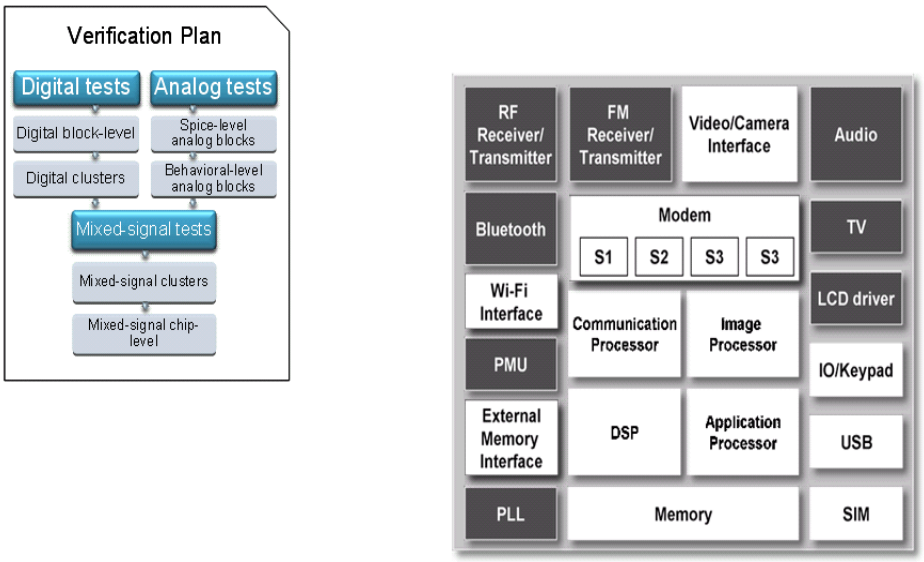


Figure 2-5 Concept engineering view

The digital block-level implementation and verification flows are mainly based on languages, such as VHDL and Verilog. Tests are created in the design language or special verification languages, such as e or SystemVerilog (See Figure 2-6).

The verification plan is updated, if needed, based on the implementation details and features of each digital block.

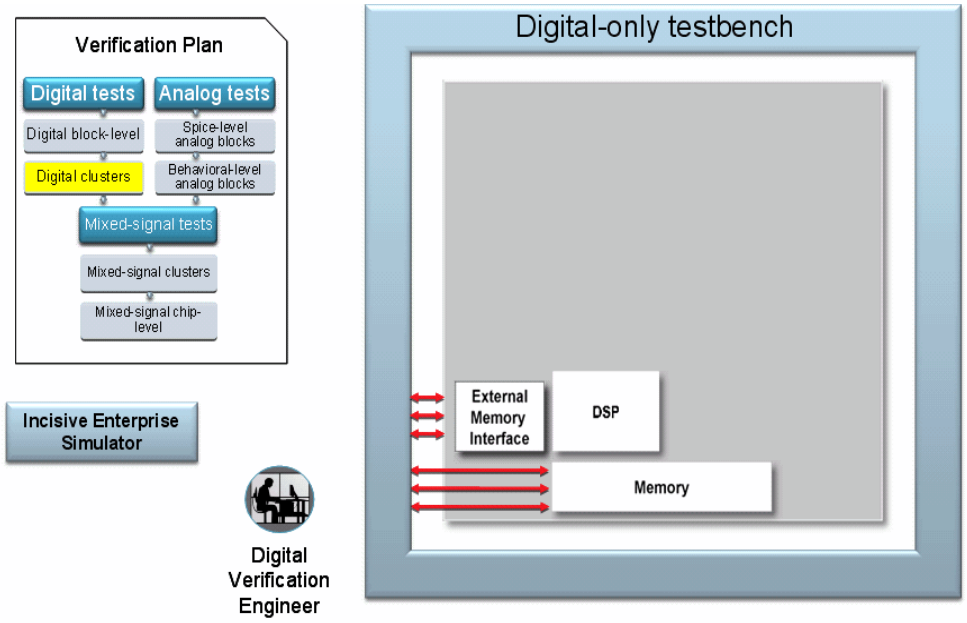


Figure 2-6 Pure digital testbench

Over time, the number of blocks and the number of tests grow (See Figure 2-7). Due to the high simulation performance, a large set of digital blocks in a complex testbench is mostly uncritical from the simulation point of view.

The pure digital verification results are captured in the Verification metric (coverage).

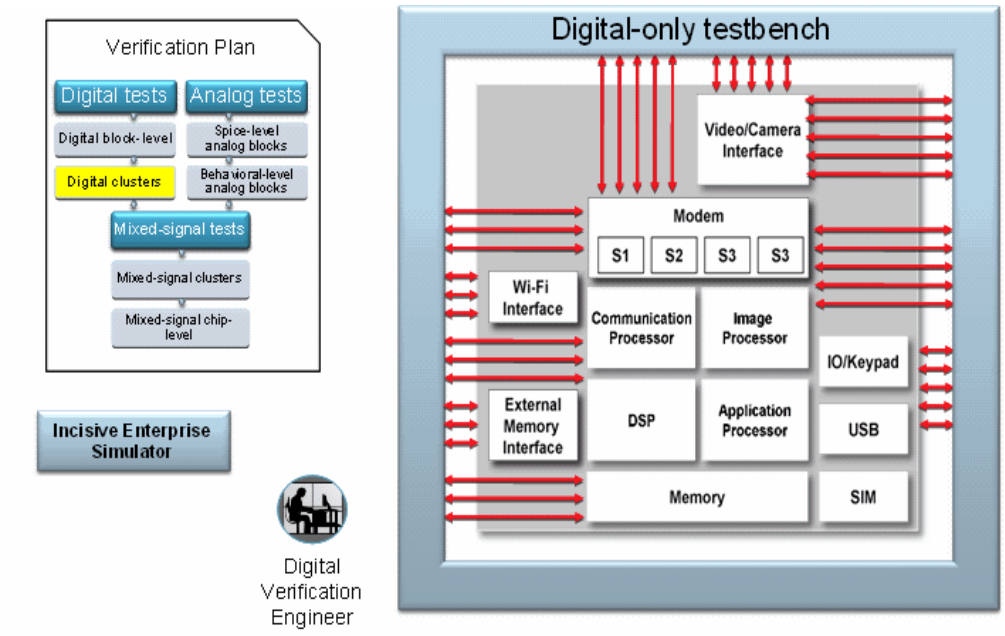


Figure 2-7 Top-level digital testbench

A similar process is followed on the pure analog implementation side (See Figure 2-8); however, the implementation and simulation environment is different. Analog users mostly use the Virtuoso Analog Design Environment (ADE) as their working environment. The testbench is a schematic. There are often different testbenches and ADE states for a single analog block. For example, for an AC configuration and a transient simulation.

Verification is often based on visual waveform inspection. Given the level of detail and size of the blocks, this is not a limiting factor for the analog designer.

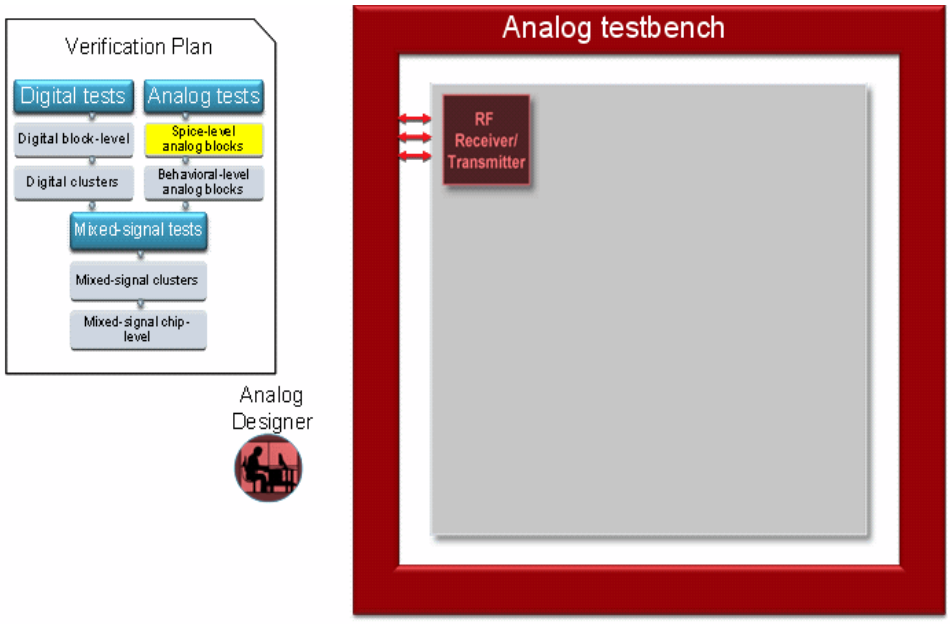


Figure 2-8 Analog block-level testbench

To improve simulation performance for top-level verification, a real value model needs to be created. This model is validated with the same analog testbench used for the transistor model to ensure whether the functionality matches the transistor-level reference or not (See Figure 2-9).

The model validation process is scheduled as a regression run to ensure that the model stays in sync with potential design changes later in the flow.

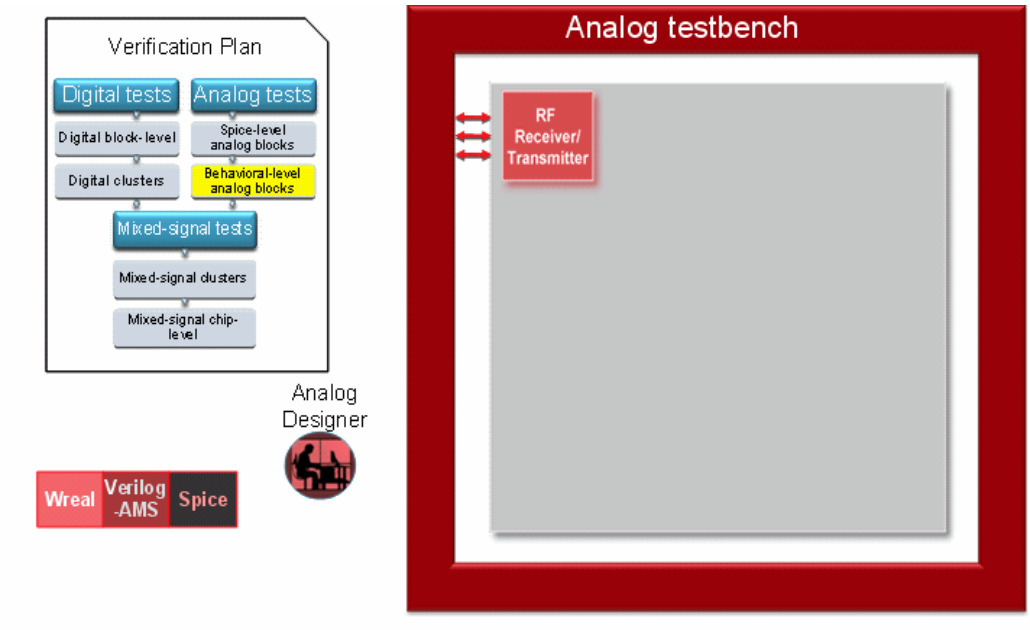


Figure 2-9 Verifying the wreal model in the analog environment

Both the analog and digital implementation flows are often mixed-signal where the analog part is represented as the transistor-level model or behavioral real model for functional verification (See Figure 2-10). The detailed analog/digital performance verification, such as impedance matching, noise/gain levels needs to be verified as completely as possible at this block-level, since the simulation performance does allow a very accurate transistor-level simulation.

This is a critical verification step in the overall chip verification task. The top-level verification based on real values is complementary to these low-level verification tasks. They are not replacing them by any means.

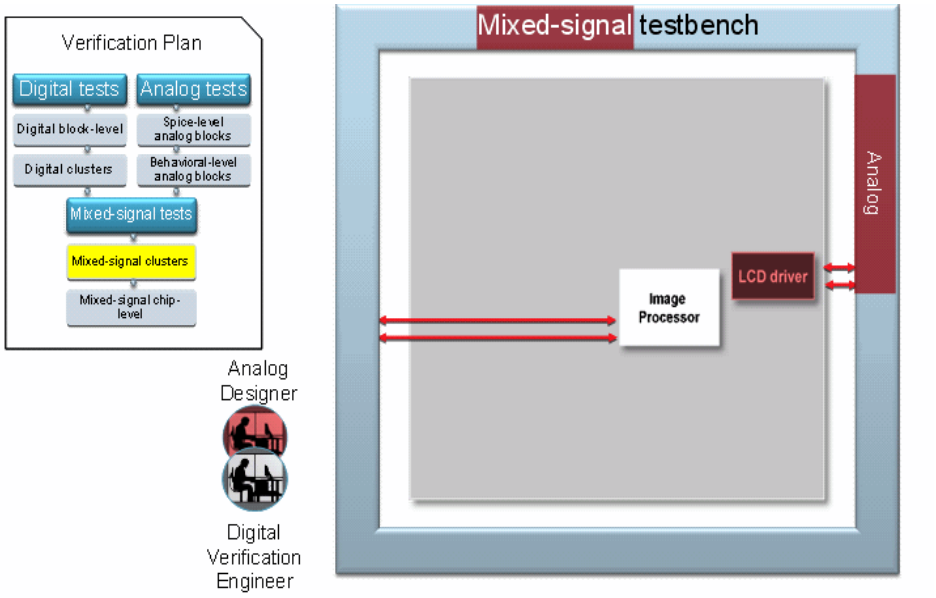


Figure 2-10 Mixed-signal block-level test

As shown in Figure 2-11 , for the top-level verification, all analog content is replaced by the real value models. This ensures a high simulation performance that is required for the top-level verification tasks.

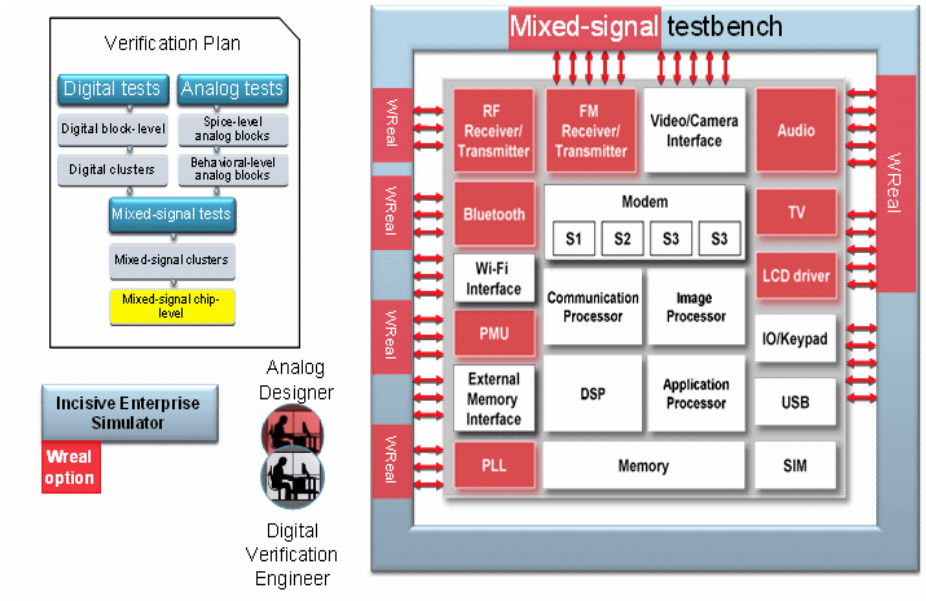


Figure 2-11 Top-level wreal testbench

Once the main functional verification is done at a high level of abstraction, it might be necessary to replace some critical blocks with the transistor-level representation again to ensure that all analog effects are correctly verified.

3

Verilog-AMS Wreal Features

Real Valued Modeling (RVM) is a method by which you can perform verification of analog or mixed-signal designs using discretely simulated real values. This allows simulation using only the digital solver, avoiding the slower analog simulation and enabling intensive verification of mixed-signal design within a short period. In this context, you need to consider the trade-off between simulation performance and accuracy. RVM also opens up the possibility of linkage with other advanced verification technologies, such as assertion-based verification, without the difficulty of interfacing with the analog engine or defining new semantics to deal with the analog values. It is anticipated that you will enable the RVM flow by migrating your analog models or transistor-level design to RVM style.

Most of the system verification in analog, digital, and mixed-signal domain is based on simulation runs. To meet the verification goals, certain amount of simulation data and data accuracy are required. For example, a detailed analysis of an RF low noise amplifier requires very high simulation accuracy, but a single RF sinusoid period might be sufficient. On the other hand, a pin connectivity check for a large digital block has an extremely low sensitivity towards accuracy, but may require a long transient simulation time to cover all sorts of events and states.

Consequently, a long full-chip simulation run using highest level of simulation accuracy would be desirable. The limiting factor in this context is simulation performance. The only practical way around this problem is a hierarchical verification approach that uses different level of design abstractions for different verification goals.

RVM is an interesting add-on to classical mixed-signal verification approaches, such as Verilog and SPICE mixed-signal simulation or pure digital modeling of the analog block in the mixed-signal design. It is not meant to replace other verification tasks including detailed analog performance verification. These tasks are still needed to ensure the correct behavior of the block. RVM complements the methodology with a high simulation performance configuration mainly targeting the functional verification goals.

Why wreal?

As mentioned above, there are several real value modeling languages. What is the advantage of wreal today?

Verilog is built on a single type – binary. Inside a module, real valued variables can be used; however, real values cannot be passed directly through ports. Passing real values through ports requires conversion to bit vectors using `$real2bits()` and `$bits2real()` system functions. This adds a lot of type conversion effort to the modeling. Moreover, the bit vector representation (for example, 64 bit vector) of the block interconnect is not matching with the physical connection (single wire).

Wreal is a native Verilog-AMS feature and includes all the benefits of a digital signal in Verilog-AMS. These are:

- Easy interaction with the analog portion of the design.
- A wreal net can be directly connected to the electrical nets by using automatically inserted E2R and R2E connect modules.
- Discipline association: The discipline concept is widely used in Verilog-AMS to check connectivity, customize connect modules, and apply resolution functions.
- Multiple wreal driver support.
- Ability for scope-based wreal driver resolution function specification.
- Identification of high impedance/unknown state(X/Z support) for real values.
- Fully supported in dfII environment, thus bridging analog and digital use models.

SystemVerilog supports the usage of real variables as ports, but there are certain limitations that apply to SystemVerilog and Verilog:

- Real value resolution functions for multiple-driver nets are not supported. It actually forbids multiple drivers for variable ports.
- No support for X/Z state
- Limited connectivity to analog models
- No Discipline association

The difference between the VHDL real and Verilog-AMS wreal capabilities are not significant. Wreal is more advanced in the area of connect modules and interfacing with the analog design portion while VHDL real is a slightly more flexible in terms of resolution function and

user defined types. In general, VHDL centric users will most likely prefer VHDL real, while Verilog users will pick Verilog-AMS wreal as the preferred language.

Verilog-AMS Wreal Language

In traditional Verilog, real values were modeled using 64-bit vectors, which encoded the real value in the IEEE floating point format. Two system tasks, $realtobits and $bitstoreal, were provided to encode and decode the real values in the 64-bit vectors. However, this use model does not support reconfigurable models under a schematic-based environment. In this environment, you model a real value with a single, scalar entity, which does not map into the traditional Verilog representation of a 64-bit vector real. This also proved difficult in the mixed-language world with VHDL reals not mapping cleanly to the 64-bit vectors used in traditional Verilog.

```
module top();
   wire [63:0] bitvector;
   source I1  (bitvector);
   sink   I2  (bitvector);
endmodule

module source(r);
   input r;
   wire [63:0] r;
   real        realnumber;
   reg [63:0]  bitvector;
   initial begin
      while (realnumber < 10.0) begin
         #1 realnumber = realnumber + 0.1;
         bitvector =  $realtobits(realnumber);
      end
      $stop;
   end
   assign r = bitvector;
endmodule // send

module sink(r);
   input r;
   real  realnumber;
   wire [63:0] r;
      always @(r) begin
      realnumber =  $bitstoreal(r) ;
```

```
      $display(" real value = %f \n", realnumber);
   end
endmodule
```

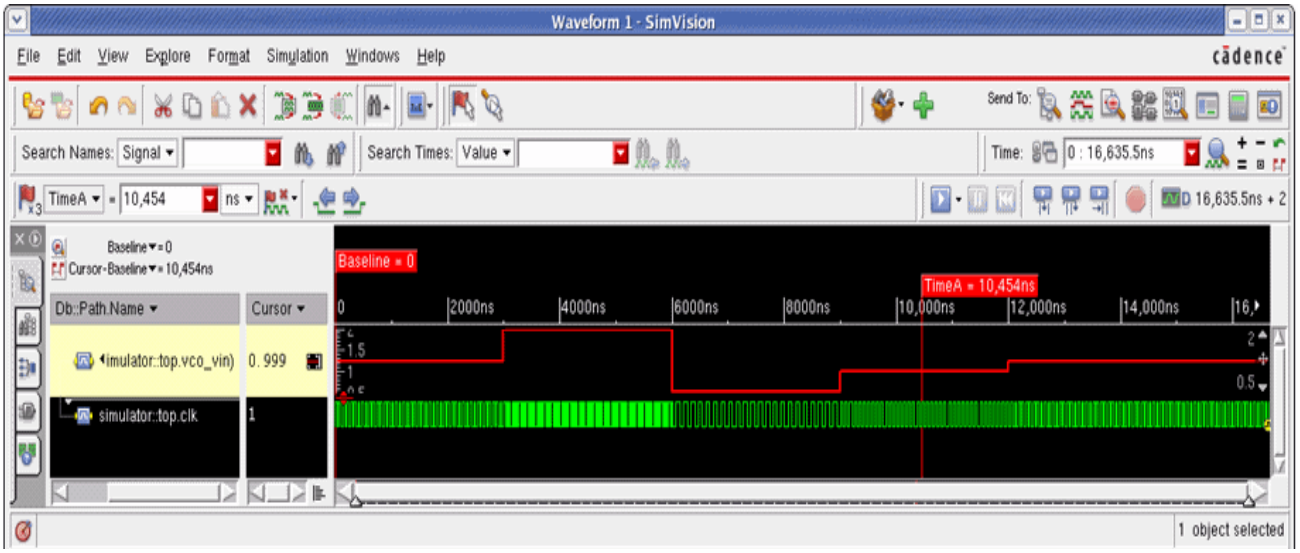
In Verilog-AMS, the concept of a truly real-valued net/wire was introduced, called "wreal" – a real valued wire. These nets represent a real-valued physical connection between structural entities. The following example illustrates the same connection between the blocks “source” and “sink” as we saw in the previous one. However, there is no type conversion needed and the connecting wire is a scalar value.

```
`include "disciplines.h"

module top();
   wreal real_wire;
   source I1  (real_wire);
   sink   I2  (real_wire);
endmodule

module source(r);
   input r;
   wreal  r;
   real  realnumber;
   initial begin
      while (realnumber < 10.0) begin
         #1 realnumber = realnumber + 0.1;
      end
      $stop;
   end
   assign r = realnumber;
endmodule // send

module sink(r);
   input r;
   wreal r;
   always @(r) begin
      $display(" real value = %f \n", r);
   end
endmodule
```

The following example illustrates the use of a real wire type as in input port of a voltage-controlled oscillator (VCO). The real value `vin` is used in an `always` block. This is possible due to the event-based nature of a wreal net—calculating the output frequency of the VCO.

```
‘include “disciplines.vams”
```

```
‘timescale 1ns / 1ps
module top ();
    wreal w in;
    real r in;
    vco vco(w in, clk);
    always begin
        r in = 1.0;
        #10 r in = 1.2;
        #10 r in = 0.2;
        #10 r in = -0.2;
        #10 r in = 1.345;
        #10 $finish;
end
assign w in = r in;
endmodule

module vco(vin, clk);
    input vin;
    wreal vin;
    output clk;
    reg clk;
    real freq, clk_delay ;
    real center freq=1; // freq in GHZ
    real vco gain=1; // freq gain in GHZ
    initial clk=0;
    always @(vin)begin
      freq = center_freq + vco_gain*vin;
      clk_delay = 1.0/(2*freq);
    end
    always #clk_delay clk = ~clk;
endmodule
```



The simulation results show the change of the output frequency according to the input values. The event-based behavior of the wreal signal is visible as well.

Wreal wires are defined as follows in the 2.3 version of the Verilog-AMS LRM (Section 3.7).

Real net declarations

The wreal, or real net data type, represents a real-valued physical connection between structural entities. A wreal net shall not store its value. A wreal net can be used for real-valued nets, which are driven by a single driver, such as a continuous assignment. If no driver is connected to a wreal net, its value shall be zero (0.0). Unlike other digital nets, which have an initial value of 'z', wreal nets shall have an initial value of zero.

Wreal nets can only be connected to compatible interconnect and other wreal or real expressions. They cannot be connected to any other wires, although connection to explicitly declared 64-bit wires can be done via system tasks $realtobits and $bitstoreal. Compatible interconnect are nets of type wire, tri, and wreal where the IEEE std 1364-2005 Verilog HDL net resolution is extended for wreal. When the two nets connected by a port are of net type wreal and wire/tri, the resulting single net will be assigned as wreal. Connection to other net types will result in an error.

The Verilog-AMS LRM lists the following restrictions on wreal nets:

- Can have at most one driver
- Can only connect to other wreals, wires, or real-valued expressions

- Scalar only and no support for arrays

The example above shows that the wreal functionality is useful for practical designs. Standard mathematical functions, such as sin, cos, abs, log, min, and max are support on real variables in standard verilog. This enables basic modeling capabilities when combining the real calculations with wreal wire connections. However, the limitations introduced by the LRM definitions have a substantial impact on the usability. As a result, Cadence extended the wreal support beyond the LRM limitations, enabling a huge variety of applications with the extended wreal features. Details are given in the next chapter.

Advanced Wreal Modeling Features

In order to provide some benefit to the wreal construct, Cadence has made significant extensions, beyond the Verilog-AMS LRM restrictions. These extensions are:

- Electrical to wreal and wreal to electrical connect modules
- Support for wreal arrays
- Support for wrealXState and wrealZState
- Support for multiple wreal drivers and resolution functions
- Support for wreal table models
- Ability to connect a wreal to a VHDL real signal or SystemVerilog real variable
- Automatic “type-casting” to wreal when a wire is hierarchically connected to a wreal, SystemVerilog real variable, or VHDL real signal

The following sections will provide a quick overview into those enhancements and the application area.

Wreal Arrays

Similar to busses, a wreal array groups multiple real values into a single, indexible entity. The following example demonstrates the definition of a wreal array:

```
module ams_tb;
    wreal y[3:2];
    sub_design d1(y);
    initial begin
    #10 $display("%f,%f",y[2],y[3]);
    end
```

```
endmodule

module sub_design(r);
    output wreal r [1:0];
    assign r[1] = 2.7182818;
    assign r[0] = 3.142818;
endmodule
```

A direct assignment of a complete array to another array is currently not supported. Assignments have to be split on the scalar level.

```
module foo(p,r);
  output wreal r [1:0];
  input  wreal p [1:0];
  // this is currently not supported
  // assign r=p;
  assign r[0]=p[0];
  assign r[1]=p[1];
endmodule
```

Wreal vectors have no useful meaning to them when considered as a single entity, thus, they are not supported.

```
    // wreal [7:0] r;
    // What would be the meaning of r as a whole?
```

However, wire vectors can legally be connected to wreal arrays, as shown in the following example.

```
module ams_tb;
   wreal y[3:2];
   wire [1:0]x;
   foo f1 (x,y);
   sub_design d1 (x);

   initial begin
      #10 $display("%f,%f",y[2],y[3]);
   end
endmodule
```

Note the wire definition in the third line. Internally, the wreal vector is converted into a wreal array of the same size.

Wreal X and Z State

The concept of an unknown – X and high impedance – Z state used in the 4-state logic is useful for wreal signals as well. The meaning of X and Z is equivalent for the wreal case. Two new keywords `wrealZState and `wrealZState are used to define the related conditions in the wreal context, as shown in the following example.

```
`include "disciplines.vams"
module top();
   wreal s;
   real r;
   foo f1 (s);
   initial begin
      #1 r = 1.234;
      #1 r = `wrealZState;
      #1 r = 3.2;
      #1 r = `wrealXState;
      #1 r = -4.2;
      #1 $stop;
   end
   assign s = r;
endmodule

module foo(a);
   inout a;
   wreal a;
   always @(a) begin
      if(a === `wrealZState)
        $display("--> Z");
      else if(a === `wrealXState)
        $display("--> X");
      else
    $display("%f", a);
   end
endmodule
```

The output result for the above example is shown below:

```
1.234000
--> Z
3.200000
--> X
-4.200000
```

`'wrealXState` and `'wrealZState` are internally defined constants. Be aware that these constants can be overwritten locally; however, this is obviously not recommended.

It is important to use the `===` operator in these type of comparisons. An `a == 1'bx` comparison always returns '`x`' due to the fact that Verilog does not assume the comparison between two unknown values to be true. Verilog is pessimistic and interprets '`x`' as `0` in the conditional statements. The `===` operator provides the comparison functionality including `x` and `z` states.

The output of the above example using the `==` instead of the `===` operator would be:

```
1.234000
`wrealZState
3.200000
`wrealXState
-4.200000
```

In all cases, the simulator would go into the `$display("%f", a);` condition.

Multiple Driven Wreals

While the resulting values of multiply driven 4-state logic nets are relatively obvious, it is not obvious what the result value on a wreal net should be if one driver provides 2.7 and another -4.87 at the same time. Should the output be the sum, the average, an X? Since there is no single answer to that question, a user-selectable resolution function is provided.

Currently (IES 82 FCS), six resolution functions are supported:

- **DEFAULT** – Single active driver only, support for Z state
- **4STATE** – Similar to Verilog 4-State resolution for digital nets
- **SUM** – Resolves to a summation of all the driver values
- **AVG** – Resolves to the average of all the driver values
- **MIN** – Resolves to the least value of all the driver values
- **MAX** – Resolves to the greatest value of all the driver values

The resolution function are selectable with `-wreal_resolution <res_func>` argument to `xmelab` or `xrun`.

Table 1 shows the results of a wreal, which is being driven by two drivers using different resolution functions:

D1	D2	Default	4state	sum	avg	min	max
x	x	x	x	x	x	x	x
x	z	x	x	x	x	x	x
x	1.1	x	x	x	x	x	x
z	z	z	z	z	z	z	z
z	1.1	1.1	1.1	1.1	1.1	1.1	1.1
2.2	1.1	x	x	3.3	1.65	1.1	2.2
1.1	1.1	x	1.1	2.2	1.1	1.1	1.1

The resolution function defined by the `-wreal_resolution` command line switch is a global setting that applies to all wreal nets in the design. Local resolution functions are supported as well; however, we need the concept of disciplines (see Discipline Naming on page 55). For more information on local resolution function, see Local Resolution Functions on page 56.

The following example illustrates the use of multiple drivers on a wreal net. The top-level wire “real_wire” is driven by the two blocks “source1” and “source2”. A third module is reading and displaying the result value of the net.

```
`include "disciplines.vams"
module top();
   wreal real_wire;
   source1 I1  (real_wire);
   source2 I2  (real_wire);
   sink    I3  (real_wire);
endmodule

module source1(r);
   output r;
   wreal  r;
   real      realnumber;
   initial begin
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealZState;
      #1 realnumber = `wrealZState;
      #1 realnumber = 2.2;
```

```
      #1 realnumber = 1.1;
      #1 $stop;
   end
   assign r = realnumber;
endmodule // send

module source2(r);
   output r;
   wreal  r;
   real      realnumber;
   initial begin
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealZState;
      #1 realnumber = 1.1;
      #1 realnumber = `wrealZState;
      #1 realnumber = 1.1;
      #1 realnumber = 1.1;
      #1 realnumber = 1.1;
      #1 $stop;
   end
   assign r = realnumber;
endmodule // send

module sink(r);
   input r;
   wreal r;
   always @(r) begin
      $display(" real value = %f", r);
   end
endmodule
```

The output of the simulation depends on the `-wreal_resolution` argument to `xrun`. For example, if the following statement is used for simulation:

```
xrun multi_driver.vams -exit -wreal_resolution sum
```

The resulting output is displayed as shown below:

```
real value = `wrealXState
real value = `wrealXState
real value = `wrealXState
real value = `wrealZState
real value = 1.100000
```

```
real value = 3.300000
real value = 2.200000
```

In the given example, the wreal drivers and receivers are nicely separated in different modules; however, the resolution function feature works the same way inside a single model. If you display `r` and `realnumber` in one of the source modules in the example, you will notice that the driven real value `realnumber` and the received wreal value `r` are different. The received wreal value represents the end result of the overall resolution process.

The resolution function provides you the ability to model analog behaviors, such as current summing nodes, in a straightforward manner.

Wreal Coercion

During the elaboration phase, the connectivity of a mixed-signal design is computed. This also involves determining and attaching types, such as logic type and electrical type to interconnects (wires). The Verilog-AMS LRM only allows wreal ports and nets to connect to wires. In this case, the wires get resolved to the type wreal. This process is called coercion to wreal.

We have enhanced this functionality in several ways. When a wire/interconnect is connected to a net of the type wreal, SV real, or VHDL real, it is coerced to wreal as well. Such coercion can occur across multiple hierarchical levels. The coercion process allows a seamless connection of devices without worrying about the interconnects and their types. This offers tremendous value in terms of model portability across various design configurations. In a different configuration, interconnect might be used to connect electrical ports – this works seamlessly without any change in the source code.

The following example illustrates the coercion function:

```
`include "disciplines.vams"

module top();
   wire w;
   sub1 I1 (w);
   sub2 I2 (w);
endmodule

module sub1(foo);
   output foo;
   source I1  (foo);
endmodule
```

```
module sub2(foo);
   input foo;
   sink   I3  (foo);
endmodule
```

We do have some levels of hierarchy and the connection between the blocks is implemented using wires. Note that we have not defined any discipline or type for the wires. This is essential to give the elaborator the flexibility to choose the appropriate wire type.

We could use the following electrical definitions for the modules `source` and `sink`:

```
`include "disciplines.vams"

module source (r);
   output r;
   electrical  r;
   analog begin
      V(r) <+ sin($abstime * 1e4);
      $bound_step(1e-5); // limit the step size
   end
endmodule // send

module sink (r);
   input r;
   electrical r;

   analog begin
      $display(" voltage = %f", V(r));
   end
endmodule
```

The entire wire hierarchy in the three top-level modules will become electrical due to the connection of the wires to the electrical ports. Using exactly the same top-level hierarchy with different leaf-level blocks will result in different wire type and discipline assignment.

```
`include "disciplines.vams"

module source(r);
   output r;
   wreal  r;
   real      realnumber;
   initial begin
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealZState;
```

```
      #1 realnumber = 2.2;
      #1 realnumber = 1.1;
      #1 $stop;
   end
   assign r = realnumber;
endmodule // send

module sink (r);
   input r;
   wreal r;
   always @(r) begin
      $display(" real value = %f", r);
   end
endmodule
```

In this case, the top-level wires are coerced to a wreal wire type. Thus, the coercion mechanism enables a straightforward reuse of testbench and sub-level hierarchies even if leaf-level blocks are swapped out by wreal blocks. Explicit definitions of wreal wire type in the upper levels of the hierarchy is not necessary.

Please note that a wreal wire that is used in any behavioral context, such as `source` and `sink` blocks should be declared explicitly as wreal type. This ensures that the object used in the behavioral code do have a well-defined type. The coercion mechanism is indented to be used for pure interconnect wires only.

Wreal Table Models

Another useful enhancement is the `$table_model` function known from analog Verilog-A and Verilog-AMS blocks. The function is now available for real values as well. Please find details of the `$table_model` function and the different options to the function in the related documentation (Cadence Verilog-AMS Language Reference).

```
`include "disciplines.vams"
`timescale 1ns / 1ps
module top();
   wreal s, vdd;
   logic out;
   real r;
   vco f1 (out, vdd, s);
   initial begin
      #10 r = 1.234;
      #10 r = 5;
```

```
      #10 r = 0.45;
      #10 r = 23;
      #10 r = 4.2;
      #1 $stop;
   end
   assign s = r;
   assign vdd = 1.1;
endmodule

module vco(out, vdd, vctrl);
   output out;
   logic out;
   input  vdd, vctrl;
   wreal  vdd, vctrl;
   reg      osc = 1'b0;
   real      tableFreq, halfper;

   always @(vdd, vctrl) begin
      tableFreq = $table_model(vdd, vctrl,"./vcoFreq.tbl",
                                          "3CC,1EL");
      halfper = 1/(2*tableFreq);
   end

   always begin
      #halfper osc = !osc;
   end

   assign out = osc;
endmodule
```

The example above calculates the VCO output frequency according to the table in the text file `vcoFrep.tbl`. This file is shown below:

```
#VDD    VCNTL   Freq/GHz
1       0       9.81
1       0.4     1.05
1       0.8     1.41
1.1     0       9.88
1.1     0.45    1.29
1.1     0.9     1.52
1.2     0       9.95
1.2     0.5     1.37
```

```
1.2     1       1.69
```

Assuming that the VDD=1.1 and VCNTRL=0.45, the output frequency would be 1.29 GHz. For values between the definition points (for example, VCNTRL=0.687), a linear or third order spline interpolation is used.

Use the `$table_model` function to model the behavior of a design by interpolating between and extrapolating outside of data points. The syntax of the table model file is identical to the `$table_model` function that have been available in the analog context for years.

```
table_model_declaration ::=
$table_model(variables , table_source [ , ctrl_string ] )

sub_ctrl_string ::=
            I
        |   D
        |   [ degree_char ] [ extrap_char [ extrap_char ]]

degree_char ::=
        1 | 2 | 3

extrap_char ::=
        C | L | S | E
```

The `ctrl_string` controls the numerical aspects of the interpolation process. It consists of sub control strings for each dimension.

When you specify I (ignore), the software ignores the corresponding dimension (column) in the data file. You might use this setting to skip over index numbers. For example, when you associate the I (ignore) value with a dimension, you must not specify a corresponding variable for that dimension.

When you specify D (discrete), the software does not use interpolation for this dimension. If the software cannot find the exact value for the dimension in the corresponding dimension in the data file, it issues an error message and the simulation stops.

degree_char is the degree of the splines used for interpolation. Important values for the degree are 1 and 3. The default value is 1, which results in a linear interpolation between data points. 3 specifies a 3rd order spline interpolation.

The extrap_char controls how the simulator evaluates a point that is outside the region of sample points included in the data file. The C (clamp) extrapolation method uses a horizontal line that passes through the nearest sample point, also called the end point, to extend the model evaluation. The L (linear) extrapolation method, which is the default method, models the extrapolation through a tangent line at the end point. The S (spline) extrapolation method

uses the polynomial for the nearest segment (the segment at the end) to evaluate a point beyond the interpolation area. The E (error) extrapolation method issues a warning when the point to be evaluated is beyond the interpolation area.

You can specify the extrapolation method to be used for each end of the sample point region. When you do not specify an extrap_char value, the linear extrapolation method is used for both ends. When you specify only one extrap_char value, the specified extrapolation method is used for both ends. When you specify two extrap_char values, the first character specifies the extrapolation method for the end with the smaller coordinate value, and the second character specifies the method for the end with the larger coordinate value.

In the example above, we used the control string ""3CC,1EL"". This specifies a 3rd order spline interpolation of the vdd variable, while the vctrl variable entries in the table are interpolated in a linear fashion. Input values for vdd below 1.0 or above 1.2 are clamped to 1.0 and 1.2 respectively. Input values below 0.0 for vctrl will result in an error, while larger value than 1.0 will be extrapolated linearly.

Connecting Wreals to Other Domains and Languages

Wreal signals need to be connected to other real type variable and signals as well as other domains, such as logical and electrical. In the same context, the connection across language, boundaries are important to discuss. How does Verilog-AMS wreal interact with VHDL and SystemVerilog for example?

Wreal connecting to VHDL real and SystemVerilog real

The connection between VHDL real and SystemVerilog real numbers is a direct connection because the data type is equivalent. The following example shows how the different languages can interact with each other. The top-level generates a real value that is passed into two sub modules, one being a VHDL and the other a SystemVerilog module.

```
`include "disciplines.vams"

module vhdl_sv_wreal ();

   wrealreal_in;
   wrealsv_real_out;
   wrealvhdl_real_out ;

   real real_in_reg;
   sv_sub i_sv_sub (real_in, sv_real_out);
   vhdl_sub i_vhdl_sub (real_in, vhdl_real_out);
```

```
   initial begin
      real_in_reg = 1.0;
      #10 real_in_reg = 5.0;
      #10 real_in_reg = 3.6;
      #10 $finish;
   end // initial begin

   always @(real_in) begin
    $display("%M: real_in = %f", real_in);
   end
   always @(vhdl_real_out) begin
    $display("%M: vhdl_real_out = %f", vhdl_real_out);
   end
   always @(sv_real_out) begin
    $display("%M: sv_real_out = %f", sv_real_out);
   end
   assign real_in = real_in_reg;

endmodule
```

In the above example, there is no interface code or any type conversion necessary to the connection. The VHDL sub module is written in pure digital VHDL using the real data type for the ports. The incoming real value is printed out to ensure that we are receiving the right value. After a multiplication by 2.0, the value is transferred to a sub module written in Verilog-AMS/wreal. The output value of the wreal sub module is again printed and multiplied before it is passed back to the top-level module.

```
library ieee;
use ieee.std_logic_1164.all;
USE STD.textio.all;
use work.all;

entity vhdl_sub is
  port (
    real_in: in real;
    vhdl_real_out : out real
    );
end;

architecture behavioral of vhdl_sub is
```

```
  signal real_2 : real;
  signal real_4 : real;

  component wreal_sub
    port (
      wreal_in: in real;
      wreal_out : out real
      );
  end component;

BEGIN
  process(real_in)
    variable l : line;
  BEGIN
    write(l, vhdl_sub'path_name);
    write(l, string'(" : real_in = "));
    write(l, real'image(real_in) );
    writeline( output, l );
  end process;

  process(real_4)
    variable l : line;
  BEGIN
    write(l, vhdl_sub'path_name);
    write(l, string'(" : real_4 = "));
    write(l, real'image(real_4) );
    writeline( output, l );
  end process;

  i_wreal_sub : wreal_sub
    port map (
      wreal_in => real_2,
      wreal_out => real_4
      );

  real_2 <= real_in * 2.0;
  vhdl_real_out <= real_4 * 2.0;
end;
```

The wreal sub block is shown below. It receives the value, prints it, and returns the double value back to the upper-level module.

```
`include "disciplines.vams"

module wreal_sub(wreal_in, wreal_out);

   input wreal_in;
   wreal wreal_in;
   output wreal_out;
   wreal wreal_out;

   real   wreal_out_reg;

   always @(wreal_in) begin
      wreal_out_reg = wreal_in * 2.0;
      $display("%M: real_in = %f", wreal_in);
   end

   assign wreal_out = wreal_out_reg;

endmodule // wreal_sub
```

In parallel to the VHDL sub module, an equivalent SystemVerilog implementation is instantiated. It performs the same operation as described above and instantiates the same sub-level wreal module.

```
module sv_sub (real_in, sv_real_out);

   input var real real_in;
   output var real sv_real_out;

   var real sv_real_2;
   var real sv_real_4;

   always @(real_in) begin
      sv_real_2 = real_in * 2.0;
      $display("%M: real_in = %f", real_in);
   end

   always @(sv_real_4) begin
      $display("%M: sv_real_4 = %f", sv_real_4);
   end

   wreal_sub i_wreal_sub (sv_real_2, sv_real_4);
```

```
   assign sv_real_out = sv_real_4 * 2.0;


endmodule // sv_real
```

As expected, the output displays the multiplication of the input value in the two sub level blocks.

```
xcelium> run
vhdl_sv_wreal:i_vhdl_sub:vhdl_sub : real_4 = 0.0
vhdl_sv_wreal:i_vhdl_sub:vhdl_sub : real_in = 0.0
vhdl_sv_wreal: real_in = 1.000000
vhdl_sv_wreal:i_vhdl_sub:vhdl_sub : real_in = 1.0
vhdl_sv_wreal.i_sv_sub: real_in = 1.000000
vhdl_sv_wreal.i_sv_sub.i_wreal_sub: real_in = 2.000000
vhdl_sv_wreal.i_vhdl_sub:i_wreal_sub: real_in = 2.000000
vhdl_sv_wreal:i_vhdl_sub:vhdl_sub : real_4 = 4.0
vhdl_sv_wreal.i_sv_sub: sv_real_4 = 4.000000
vhdl_sv_wreal: sv_real_out = 8.000000
vhdl_sv_wreal: vhdl_real_out = 8.000000
```

Connection to the Electrical Domain

Similar to the mechanism of automatically inserted connect modules (AICM) between the electrical and logic domain, these connect modules are available between wreal and electrical and vice versa. The connect modules (CM) are called E2R, R2E_2, and ER_bidir. They are inserted automatically, if needed, during the elaboration process.

The current connect modules is shown below. The source file can be found in the IUS/IES release in <install_path>/tools/affirma_ams/etc/connect_lib/. The Real to Electrical (R2E) CM implementation is relatively straightforward. The newly introduced X and Z states are taken into account. The input wreal value is assigned to a voltage source with a serial resistance. Transition operators help avoid abrupt changes in the behavior that might convergence issues in the analog solver.

```
// R2E_2.vams - Efficient Verilog-AMS discrete wreal to
// electrical connection module

`include "disciplines.vams"
`timescale 1ns / 100ps

connectmodule R2E_2 (Din, Aout);
  input Din;
```

```
  wreal Din;// input wreal
  \logic Din;
  output Aout;
  electrical Aout;// output electrical

  parameter real vsup = 1.8from (0:inf);
                    // supply voltage
  parameter real vdelta = vsup/64 from (0:vsup];
                    // voltage delta
  parameter real vx = 0 from [0:vsup];
                    // X output voltage
  parameter real vz = vxfrom [0:vsup];
                    // Z output voltage
  parameter real tr = 10pfrom (0:inf);
                    // risetime of analog output
  parameter real tf = trfrom (0:inf);
                  // falltime of analog output
  parameter real ttol_t = (tr+tf)/20 from (0:inf);
                   // time tol of transition
  parameter real tdelay = 0from [0:inf);
// delay time of analog output
  parameter real rout = 200from (0:inf);
// output resistance
  parameter real rx = routfrom (0:inf);
// X output resistance
  parameter real rz = 10Mfrom (0:inf);
// Z output resistance

real Vstate, Rstate;
  real Vout, Rout;
  initial begin
    if (Din === `wrealXState)
      begin Vstate = vx; Rstate = rx; end
    else if (Din === `wrealZState)
      begin Vstate = vz; Rstate = rz; end
    else
      begin Vstate = Din; Rstate = rout; end
  end
  always @(Din) begin
    if (Din === `wrealXState)
      begin Vstate = vx; Rstate = rx; end
```

```
    else if (Din === `wrealZState)
      begin Rstate = rz; end
    else if (Din-Vstate >= vdelta || Vstate-Din >= vdelta)
      begin Vstate = Din; Rstate = rout; end
  end
  assign Din = Din;
  analog begin
    Vout = transition(Vstate, tdelay, tr, tf, ttol_t);
    Rout = transition(Rstate, tdelay, tr, tf, ttol_t);
    I(Aout) <+ (V(Aout) - Vout) / Rout;
  end
endmodule
```

In contrast to the R2E, the E2R (Electrical to Real) operation is not so obvious because the continuous analog value needs to be translated into an event-based wreal signal. For this purpose, we created a new system function, `absdelta`, which creates events based on input value changes. In the CM below, the `absdelta` function issues an event and thus updates the wreal value whenever V(`Ain`) changes more than the value `vdelta`.

More precisely, the `absdelta` function generates events for the following times and conditions

```
(absdelta ( expr, delta [ , time_tol [ , expr_tol ]]):
```

- At time zero
- When the analog solver finds a stable solution during initialization
- When the expr value changes more than delta plus or minus expr_tol, relative to the previous `absdelta` event (but not when the current time is within time_tol of the previous absdelta event)
- When expr changes direction (but not when the amount of the change is less than expr_tol

```
// E2R.vams - basic Verilog-AMS electrical to discrete wreal connection module
// last revised:  08/01/06, jhou
//
// REVISION HISTORY:
// Created:  08/01/06, jhou

`include "disciplines.vams"
`timescale 1ns / 100ps

connectmodule E2R (Ain, Dout);
```

```
input Ain;
electrical Ain;    //input electrical
output Dout;
wreal Dout;        //output wreal
\logic Dout;       //discrete domain

  parameter real vdelta=1.8/64   from (0:inf);
// voltage delta
  parameter real vtol=vdelta/4   from (0:vdelta);
// voltage tolerance
  parameter real ttol=10p        from (0:1m];
// time tolerance

  real Dreg;       //real register for A to D wreal conversion

  assign Dout = Dreg;

//discretize V(Ain) triggered by absdelta function
  always @(absdelta(V(Ain), vdelta, ttol, vtol))
    Dreg = V(Ain);
endmodule
```

The proper setting of the CM parameters is critical for correct simulation results. While the supply voltage level mainly influences the electrical to logic connection thresholds, the E2R settings can be totally independent of the supply voltage. For example, if we pass an electrical 5 mV bias voltage into a wreal block, a good setting for a `vdelta` parameter might be 100uV, independent of what the supply voltage is.

Also, consider the default resistance in the R2E CM. If a wreal net is connected to an electrical part that consumes significant current (for example, power net) the 200-Ohm default resistance of the R2E CM might have an unwanted influence on the signal level.

We will talk about ways to influence the CM insertion by discipline settings in later sections. The following example shows a simple connection between electrical and wreal. The E2R connect module will be inserted automatically. The connect rule definition at the end of the file defines the parameters for the connect modules.

```
// run with:
// xrun cm.vams cm.scs -clean -amsconnrules e2r_only
//        -discipline logic
`include "disciplines.vams"
`timescale 1ns / 1ps
module top();
```

```
   wreal real_wire;
   source I1  (real_wire);
   sink   I3  (real_wire);
endmodule

module source(r);
   output r;
   electrical  r;
   analog begin
      V(r) <+ sin($abstime * 1E4);
      $bound_step(1e-5); // limit the step size
   end
endmodule // source

module sink(r);
   input r;
   wreal r;
   always @(r) begin
      $display(" real value = %f", r);
   end
endmodule
connectrules e2r_only;
  connect E2R
      #( .vdelta(0.1), .vtol(0.001), .ttol(1n));
Endconnectrules
```

The `analog.scs` file contains the following statements:

```
simulator lang=spectre
tran1 tran stop=1ms
```

Connection to the Digital Domain

The concept of automatic inserted connect modules applies between the discrete (logic and wreal) and continuous, such as electrical and mechanical domain. It also applies to the connection between two discrete domains, such as wreal to logic connection. The connect modules (CM) are called L2R and R2L. They are inserted automatically, if needed, during the elaboration process.

The following is an example of a basic logic to real (L2R) connect module:

```
connectmodule L2R(L, R);
```

```
    input L; \logic L;
    output R; wreal_dsp R;
    parameter real vsup = 1.8 from (0:inf);
    parameter real vlo = 0;
    parameter real vhi = vsup from (vlo:vsup);
    parameter real vtlo = vsup / 3;
    parameter real vthi = vsup /1.5;

    wire [31:0] L_val;
    reg [1:0] L_code;
    real L_real;

    initial
        begin
            $BIE_input_strength(L, L_val);
    end

    // Determine the value and strength of L and convert to a real number

    always
        begin
            L_code = L_val & 2'b11;
            case (L_code)
            2'b00: L_real = vlo;
            2'b01: L_real = vsup;
            2'b11: L_real = `wrealXState
            2'b10: L_real = `wrealZState;
        endcase

        @(L_val)

    end

    // drive the converted value back onto the R output pin

    assign R = L_real;

endmodule
```

The following is an example of the R2L connect module:

```
connectmodule R2L(L, R);
    output L; \logic L;
    input R; wreal_dsp R;
    parameter real vsup = 1.8 from (0:inf);
    parameter real vlo = 0;
    parameter real vhi = vsup from (vlo:vsup);
    parameter real vtlo = vsup / 3;
    parameter real vthi = vsup /1.5;

    wreal R_val;
    reg R_logic;

    initial
        begin
            $BIE_input_real(R, R_val);
    end

    // Determine the value of R and convert to a logic value

    always
        begin
            if(R_val >= vthi)
                R_logic = 1'b1;
            else if (R_val <= vtlo)
                R_logic = 1'b0;
            else if(R_val === `wrealZState)
```

```
                R_logic = 1'bz;
            else
                    R_logic = 1'bx;
            @(R_val);
        end
    // drive the converted value back onto the output L pin
    assign L = R_logic;
endmodule
```

Working with Disciplines

The concept of disciplines was introduced in the Verilog-AMS language to differentiate different domains within discrete and continuous design part. For example, disciplines are used to separate different power domains, such as a 3.3 V logic and a 1.8 V logic region. The same applies for the continuous domain. While the systems of equations for electrical and mechanical descriptions are identical for the simulation, the disciplines are used to apply different settings like access functions to both design parts.

Elaborator and simulation functions are influenced by the discipline settings. Also, the connect modules to be used in certain places are controlled by the discipline.

The discipline definition could be applied within the design itself or it can be assigned to design objects by using the `-setdiscipline` option of `xrun/xmelab`. The later choice provides a powerful mechanism to influence the simulator without changing the underlying design information. For the below example, the command line setting is shown as follows:

```
-setd "INSTTERM-top.I1.UP- wreal_voltage"
-setd "INSTTERM-top.I1.DN- wreal_voltage"
-setd "INSTTERM-top.I1.Z- wreal_current".
```

The following example introduces the concept of connect module insertion being controlled by the discipline definition. A wreal model of a very simple charge pump is used inside an analog testbench. The stimuli are created by pulse voltage sources. The output of the charge pump is connected to a simple RC filter.

The charge pump circuit drives current into the RC filter according to the input voltage levels. However, simulating this design as it would not lead to the expected simulation results. The wreal output signal of the charge pump is +/- 300u. This value is far smaller than the normal accuracy setting for R2E CM, that might be in a range of 1m. Moreover, the normal R2E CM is an electrical voltage source at the output. This does not fit to our requirements here.

```
module top();
   electrical in1, in2, out, gnd;
   ground gnd;
```

```
   vsource #(.type("pulse"), .val0(0), .val1(3.3), .period(2n),
             .delay(0), .rise(100p), .fall(100p), .width(1n))
           V1 (in1, gnd);
   vsource #(.type("pulse"), .val0(0), .val1(3.3),
             .period(1.87n), .delay(0.5n), .rise(100p),
             .fall(100p), .width(1n))
           V2 (in2, gnd);

   CP I1  (in1, in2, out);

   capacitor #(.c(4p)) c1 (out, gnd);
   resistor #(.r(20k)) r1 (out, gnd);
endmodule

module CP(UP, DN, Z );
   input UP;                 // Increment Input Controls
   input DN;                 // Decrement Input Controls
   output Z;                 // Single ended output

   wreal UP, DN, Z;
   wreal_current Z;        // Disciplines could be hardcoded
   wreal_voltage UP, DN;  //   or set with the -setd option
   parameter real I_out    = 300u;           // Output Current
   parameter real thres    = 1.5;            // threshold value
   real      iup, idn, out;

   always @(UP) begin
      if (UP > thres )
    iup = I_out;
      else
    iup = 0.0;
      out = iup - idn;
   end

   always @(DN) begin
      if (DN > thres )
    idn = I_out;
      else
    idn = 0.0;
      out = iup - idn;
   end
```

```
   assign Z = out;
endmodule
```

A few things need to be done to setup the testcase correctly. First, we need to define a wreal to electrical connect module with a current source output, as shown below. To simplify matters only basic conversion features have been taken into account. However, `wrealX/ZStates are not considered.

The standard E2R CM can be used for the voltage to wreal conversion at the input of the charge pump.

```
connectmodule R2E_current (Din, Aout);
   input Din;
   wreal Din;   // input wreal
   \logic Din;
   output Aout;
   electrical Aout;// output electrical


parameter real tr = 10p from (0:inf);
// risetime of analog output
   parameter real tf = tr   from (0:inf);
// falltime of analog output
   parameter real ttol_t = (tr+tf)/20   from (0:inf);
// time tol of transition
   parameter real rout = 1M from (0:inf);
// output resistance
   real      Iout;

   assign Din = Din;
   analog begin
      Iout = transition(Din, 0, tr, tf, ttol_t);
      I(Aout) <+ (V(Aout) / rout) - Iout;
   end
endmodule
```

As said earlier, discipline settings are used to control the connect module insertion process. For this purpose, we need to specify two new discrete disciplines for the “voltage” and the “current” domain inside our wreal model.

```
discipline wreal_current
  domain discrete;
enddiscipline

discipline wreal_voltage
```

```
  domain discrete;
enddiscipline
```

The next step is to associate the connect modules with the appropriate discipline interfaces.

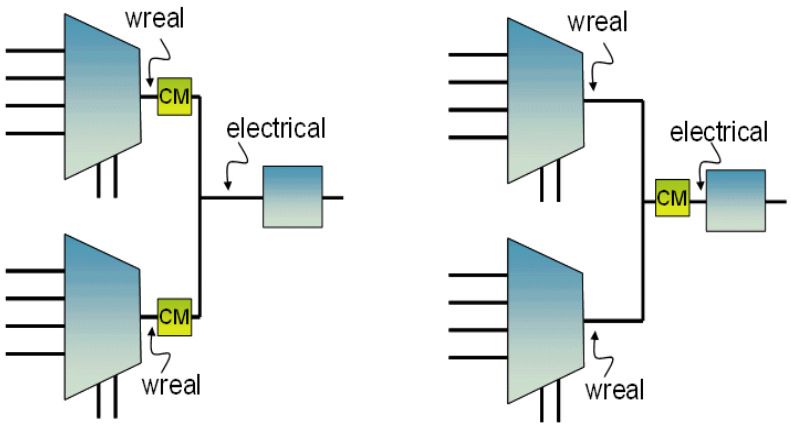
```
connectrules wreal_V_I;
  connect E2R
      #( .vdelta(0.1), .vtol(0.001), .ttol(1n))
      electrical, wreal_voltage;
  connect R2E_current
      #( .tr(0.1n), .rout(1M) )
      wreal_current, electrical;
endconnectrules
```

The standard E2R/R2E connect module is used to combine the logic and electrical discipline. In this case, we need a connect module between the electrical and the `wreal_voltage` discipline. The default settings can be overwritten in the connect rules definition as shown above. The same applies for the R2E connect module. In addition, we use the newly defined R2E_current connect module instead of the default one.

Note that the defined current connect module is oversimplified to understand basic principles. For real usage, we need to consider more details, such as E2R_current. The current is generated from the global ground node. Thus, it is not considered in the power nets. The current source is not supply-limited that means it would create current even if the voltage is above or below the supply range.

Current based connect modules are sensitive about the placement of the CMs in the design. See the figure below. The output current of the one/two connect modules would be quite different in the two cases (assuming a wreal resolution function other than sum). To avoid these type of problems, current based CM are recommended for point-to-point connections

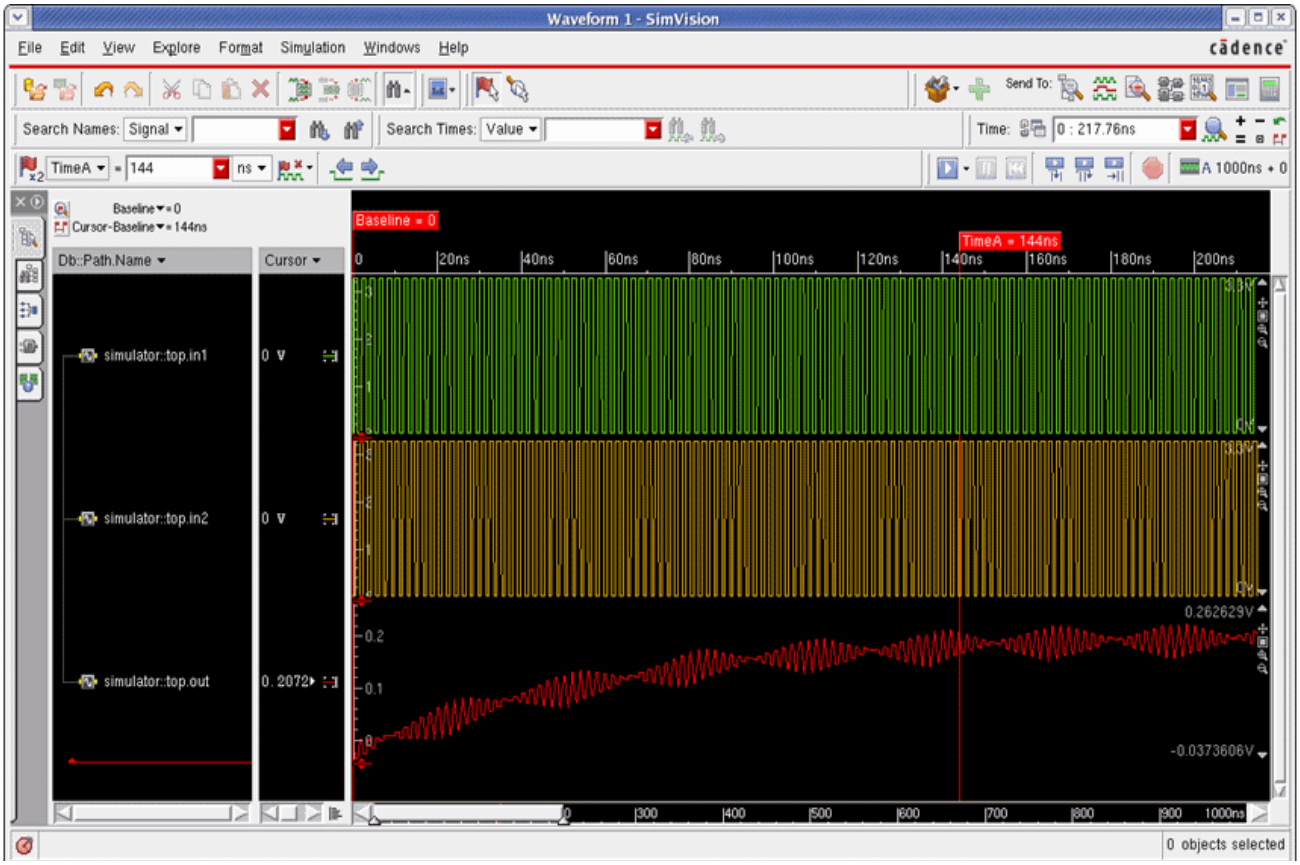
only or you have to make sure that the resolution function sum is used for the wreal net in question.



The final step is the definition of the disciplines in the design. This is done as hard coded assignments, as shown in the two commented lines in the charge pump definitions. The source code modification is often not appropriate. In these cases, the disciplines can be set by `-setdiscipline` options to `xrun/xmelab`. These external settings allow discipline definitions without any source code modifications. There are different scopes available for the `setdiscipline` option, including library, cell, and instances. For the example shown above, the following command line `run` option would be appropriate. All the code examples and the discipline.vams and timescale settings are assumed to be included in the `cp.vams` file.

```
xrun cp.vams analog.scs -clean -amsconnrules wreal_V_I -setd "INSTTERM-top.I1.UP-
wreal_voltage" -setd "INSTTERM-top.I1.DN- wreal_voltage" -setd "INSTTERM-top.I1.Z-
wreal_current"
```

The figure below shows the simulation results for the example. The loading of the capacitor through the charge pump output current is clearly visible.



Discipline Naming

Discipline names are user-defined strings. The basic disciplines are defined in `<IUS/IES path>/tools/spectre/etc/ahdl/disciplines.vams`. This includes the default discipline “logic” for the discrete part and various continuous disciplines, such as electrical, magnetic, thermal, and translational.

```
discipline my_electrical
    potential    Voltage;
    flow         Current;
enddiscipline

discipline my_logic
  domain discrete;
enddiscipline
```

New disciplines can also be defined as shown in the examples above. Continuous disciplines need the potential and flow definition while the discrete discipline is defined only with the keyword `discrete`.

There is no standard available for variable names other than the default definitions. The following list recommends a useful naming convention for the discipline definition. Appling these naming conventions simplifies the IP exchange between different groups.

If different discipline naming had been used, the `connect` or `resolveto` command defines identical/compatible disciplines. In the below example, disciplines A, B, and C are defined as compatible. They are all resolved toward the discipline A.

```
connectrules cr;
  connect  A, B, C resolveto A;
endconnectrules
```

Here is a list of recommended names for wreal disciplines.

- wreal_voltage normal voltage:
  R2E has default output resistance.
- wreal_vsup power supply:
  R2E has zero output resistance.
- wreal_vmatch matched load
  R2E is driven with twice the wreal value through output resistor Rmatch. E2R input resistance is also Rmatch.
- wreal_current ideal current
  R2E is current with impedance. E2R is a fixed-bias voltage plus load resistance. Supply limiters in R2E could prevent that the voltage exceeds gnd or vdd levels.
- wreal_iloadgnd gnd based current
  R2E is resistor from pin to ground, no E2R.
- wreal_iloadsup supply based current
  R2E is resistor from pin to supply, no E2R.

Local Resolution Functions

We introduced the concept of wreal resolution functions above; however, the setting was limited to one global resolution function. This is not appropriate in larger designs where wreals are used to model different types of interconnects (For example, voltage and currents).

To differentiate which resolution function should be used for which net, the discipline concept introduced above is used. Starting in IUS/IES82_s010 disciplines definitions can contain

realresolve statements that control the resolution function for the discipline. Moving forward we will simplify the use model significantly hiding some of the discipline definition overhead that is required today. The following code example defines a sum resolution function for the wreal_current discipline.

```
discipline wreal_current
  domain discrete;
  realresolve  sum;
enddiscipline
```

Consider the same example as above for the resolution function. Two drivers are stimulating the same net and a receiver displays the resolution values. In this case, we have two instances of all three blocks in the top-level and we want to control the resolution function separately for both.

We are using the discrete disciplines `wreal_current` and `wreal_voltage` for the two wreal wires. The discipline and the related resolution functions are defined first. The `wreal_current` discipline is using the sum resolution function assuming that current summing is the appropriate behavior in this case. The voltage related wreal discipline is using the average function that provides the right output value under the assumption that the driver strengths are equivalent.

```
`include "disciplines.vams"
`timescale 1ns/1ps

discipline wreal_current
  domain discrete;
  realresolve  sum;
enddiscipline

discipline wreal_voltage
  domain discrete;
  realresolve  avg;
enddiscipline

module top();
   wreal real_wire1;
   wreal_current  real_wire1;
   source1 I11  (real_wire1);
   source2 I21  (real_wire1);
   sink   I31  (real_wire1);
   wreal real_wire2;
   wreal_voltage real_wire2;
   source1 I12  (real_wire2);
```

```
   source2 I22  (real_wire2);
   sink   I32  (real_wire2);
endmodule

module source1(r);
   output r;
   wreal  r;
   real      realnumber;
   initial begin
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealZState;
      #1 realnumber = `wrealZState;
      #1 realnumber = 2.2;
      #1 realnumber = 1.1;
      #1 $stop;
   end
   assign r = realnumber;
endmodule // send

module source2(r);
   output r;
   wreal  r;
   real      realnumber;
   initial begin
      #1 realnumber = `wrealXState;
      #1 realnumber = `wrealZState;
      #1 realnumber = 1.1;
      #1 realnumber = `wrealZState;
      #1 realnumber = 1.1;
      #1 realnumber = 1.1;
      #1 realnumber = 1.1;
      #1 $stop;
   end
   assign r = realnumber;
endmodule // send

module sink(r);
   input r;
   wreal r;
```

```
   always @(r) begin
      $display("%m --> real value @ %f = %f", $abstime/1n, r);
   end
endmodule
```

It is not appropriate in all cases to hardcode the resolution functions and discipline assignment in the Verilog-AMS source. We have discussed about assigning the discipline using the `-setdiscipline` switch in the previous section. The resolution function itself can also be applied to a discipline during the elaboration time. If we define an ams control file (using the file extension `*.scs`) with the following content:

```
amsd {
  connectmap discipline="wreal_current" realresolve=sum
  connectmap discipline="wreal_voltage" realresolve=avg
}
```

The resolution function assignment would give the same result as defined in the hard coded example above. The `xrun` switch `+wreal_res_info` provides information of the resolution function used for net in the design.

4

Modeling with Wreal

In this section, we will focus on wreal modeling examples to illustrate the modeling process. Due to the different modeling style in the electrical domain and in wreal, it is impossible to provide a 1 to 1 mapping between all functions and modeling practices. It is also impossible to provide a “recipe” on how to create models. The best starting point is an example model that does something similar to what you need. By modifying and extending the example, you will build up enough experience to create more sophisticated wreal models. The example discussed below will help you getting a good starting point for your modeling experiences.

Sample Model Library

Cadence provides a sample model library of wreal models. Some of the example in this book can be found in the sample library. The entire source code of the library is accessible making it an excellent vehicle for further education.

The sample library can be found in your IUS/IES installation directory in `$IUSHOME/tools/amsd/wrealSamples/`. We provide three versions of the sample library, a pure text based version for command line users, a IC5141 dfII version, and a IC61x dfII version. Command line users can directly instantiate the models. Environment users have to extend their `cds.lib` file to point to the appropriate library location.

First Example: Voltage Controlled Oscillator

Consider the example of voltage-controlled oscillator (VCO). VCO is a standard analog block that is used for examples in PLLs. It creates an output oscillation with a given frequency. This frequency can be modified to some range by the input voltage of the block.

The testbench is shown below. A real variable (`r_in`) is changed every 10 ns from one value to another. The real variable is assigned to the wreal wire (`w_in`). This wreal signal is connected to the input port of the VCO block that is instantiated in the testbench.

```
`include "disciplines.vams"
`timescale 1ns / 1ps
```

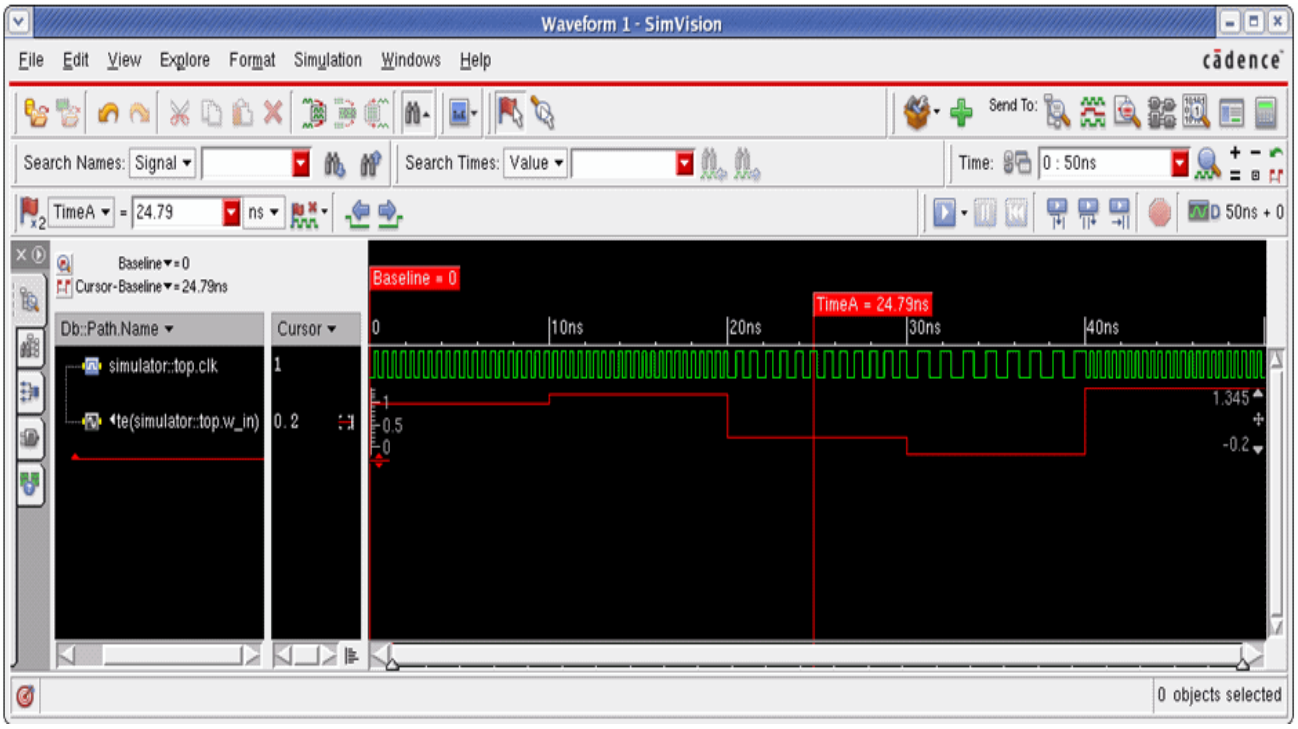
```
module top();
   wreal w_in;
   real r_in;
   vco vco (w_in, clk);

   always begin
      r_in = 1.0;
      #10 r_in = 1.2;
      #10 r_in = 0.2;
      #10 r_in = -0.2;
      #10 r_in = 1.345;
      #10 $finish;
   end
   assign w_in = r_in;
endmodule
```

The VCO block takes the wreal input signal and calculates at each change of the signal (`@(vin)`) the required output frequency. Remember that the wreal signal is event based. That means that there are discrete events whenever the signal changes and the signal stay constant otherwise.

The calculated output frequency is used to determine the delay time between the clock inversion operations. Please note that the frequency settings depend on the timescale used. In this case, we used a 1 ns timescale resulting in a 1 GHz frequency unit.

```
module vco(vin, clk);
   input vin;
   wreal vin;
   output clk;
   reg  clk;
   real   freq, clk_delay;
   real   center_freq = 1;  // freq in GHz
   real   vco_gain = 1;     // freq gain in GHz

   initial clk = 0;

   always @(vin) begin
      freq = center_freq + vco_gain*vin;
      clk_delay = 1.0/(2*freq);
   end
   always #(clk_delay)  clk = !clk;
endmodule
```

The following figure shows the simulation result for this example.



It is obvious that this example is oversimplified. We have not used any parameters with value range restrictions for the constant definitions, input values of type `` `wrealXState `` and `` `wrealZState `` are not considered. It is possible to create negative frequency values and thus negative delay times, the clock frequency changes instantaneously with the input voltage change. All these – and probably more – issues and effects need to be considered for a good, robust, and reusable behavioral model. We will address some of these in later examples.

Analog Functions Translated to Wreal

Some analog operations can be implemented in wreal in a very straightforward manner, others need a real translation. In the following section, we will discuss some examples of standard tasks that are needed during wreal modeling. We are mostly using example code to highlight the modeling style.

Wreal Value Sources

How can we generate sources with values varying over time? The following example is a source for an analog amplifier that we will discuss later on. It creates a differential signal on the output ports P and N.

```
`include "constants.vams"
`include "disciplines.vams"
`timescale 1ps/1ps

module wrealAmp_stim (P,N);
output P,N;
wreal P,N;

real Vin;                  // input voltage
real Freq=600M,Phase=0;    // sinusoid params

initial begin  // drive input, comment on expected result:
  Vin=0.1;          // out=1 for DC op point
  #50  Vin=0.108;   // out=1.08 in <20ps
  #100 Vin=0.15;    // out=1.5  in 100ps
  #150 Vin=0.5;     // out sat=3.0  in 300ps
  #400 Vin=0;       // out=0    in 600ps
  #700 Vin=0.2;     // out=2.0 after 400ps, but:
  #300 Vin=0.1;     // prior to full change, ramp down to out=1
  #300 while ($abstime<6000p) begin
                                // generate ramped sine input
    #20 Freq=Freq*1.007;        // gradual freq increase
    Phase=Phase+20p*Freq;       // integrate freq to get phase
    if (Phase>1) Phase=Phase-1; // wraparound per cycle
    Vin=0.1*(1+sin(`M_TWO_PI*Phase));
                                // sinusoidal waveform shape
  end
  #200 $finish;                 // done with test
end

assign P = Vin/2;    // drive symmetric diffl signal to inputs
assign N = -Vin/2;
endmodule
```

A real value (`Vin`) is used to assign different values to it after certain delays. Discrete events are created on the real value, for example, at 150 ps the value is changing from 0.108 to 0.15. At 300 ps, a sinusoidal signal is generated. The loop is updated every 20 ps with a new value generated by the sin function. Note that we need to define the sampling time for these types of continuous signals. Wreal is changing at discrete events, thus, a continuous signal need to be approximated by a given sampling rate. The appropriate sampling rate and the decision on using a fixed or a flexible time step depends on the sampled signal and the accuracy requirements for the following blocks.

The real value is assigned to the wreal outputs P and N is a symmetrical way.

Integration and Differentiation

The following example shows an analog integration and differentiation on a given sinusoid input function.

```
`include "disciplines.vams"
`timescale 1ns/1ns

module top ();
   electrical x, idt_x, ddt_x;

   w_idt I_w_idt (x);
   w_ddt I_w_ddt (x);

   analog begin
      V(x) <+ sin($abstime*1E7);

      V(ddt_x) <+ ddt(V(x));
      V(idt_x) <+ idt(V(x),0);
   end
endmodule // top
```

The same input waveform is converted into a wreal signal using the E2R connect rules with a 0.1 V accuracy.

```
connectrules e2r_only;
  connect E2R
      #( .vdelta(0.1), .vtol(0.001), .ttol(1n));
endconnectrules
```

Two sub modules implement the integration and differentiation function in the discrete domain. The only information needed in addition to the input signal is the time point of the last event (`lasttime`) and the value at this time (`lastval`). The time difference between the last event and the current time determines the integration interval for this particular step. A simple multiplication with the last wreal input increments the integral values. The initial condition for the integral is set to 0.0.

```
module w_idt(w_x);
   input w_x;
   wreal w_x, w_idt_x;
   real  r_idt_x  = 0;
   real  lasttime = 1;
```

```
   real  lastval  = 0;

   always begin
      if (lasttime < $abstime) begin
         r_idt_x = r_idt_x + lastval * ($abstime - lasttime);
      end
      lasttime = $abstime;
      lastval = w_x;
      @(w_x);
   end
   assign w_idt_x = r_idt_x;
endmodule
```

The differentiation operator is implemented very similar. Dividing value and time difference gives the derivative of the last step. Please note that this is the linear interpolated derivative over the given time period.

```
module w_ddt(w_x);
   input w_x;
   wreal w_x, w_ddt_x;
   real  r_ddt_x  = 0;
   real  lasttime = 1;
   real  lastval  = 0;

   always begin
      if (lasttime < $abstime) begin
        r_ddt_x = (w_x - lastval) / ($abstime - lasttime);
      end
      lasttime = $abstime;
      lastval = w_x;
      @(w_x);
   end
   assign w_ddt_x = r_ddt_x;
endmodule
```

It should be considered that the step size of the wreal events highly influences the derivative calculation. Large steps can lead to inaccurate values. Moreover, the differentiation and integration function will update only when a wreal event occurs. For example, if you were calculating the integral of a constant value you would not get any results because there is only one event on the wreal signal at time point zero. In such cases, you need to sample the input signal appropriately, for example, by the fix rate sampling shown above.

The `w_idt` function implements the integral of the wreal signals sample and hold behavior while the `w_ddt` assumes a linear interpolation. It is a matter of which definition concept you follow. Generally, the sample and hold behavior is more digital-like while the linear interpolation is closer to the analog signal nature. If you want to integrate wreal as linear interpolated signal, a trapezoidal integration of the values would be more appropriate. This would result in:

```
r_idt_x = r_idt_x +
          0.5 * (w_x + lastval) * ($abstime - lasttime);
```

The differentiation of the sample and hold interpretation of the signal results in a Dirac pulse, which is not useful for most models.

Value Sampling

Wreal signals are event-based. They can have a fix sampling rate or the step size from one event to another can vary from step to step. During the conversion from a continuous domain into the discrete wreal domain, events are created based on the amount of value change (see above). Even inside the event-based simulation, it is often necessary to change the sampling rate from one block to another.

The following example shows a triangular step wave input signal that should be sampled at a lower rate. We implemented two different sample modules to show the different behavior.

```
//xrun sampler.vams -gui -access r
`include "constants.vams"
`include "disciplines.vams"
`timescale 1ns / 1ps
module top ( s_out );

   output s_out;
   wreal  s_out, s_sample, s_sample_simple;
   real   r_out;

   fix_rate_sampler #(.sr(200M)) I_frs (s_out, s_sample);
   fix_rate_sampler_simple #(.sr(200M)) I_frss (s_out, s_sample_simple);

   always begin
      repeat (5) begin
     r_out = 0.0;
     #1 r_out = 1.0;
     #1 r_out = 2.0;
     #1 r_out = 3.0;
```

```
      #1 r_out = 4.0;
    #1;
     end
     $finish;
  end
  assign s_out = r_out;

endmodule
```

The first is a very simple sampling module that is triggered at the given sampling rate. At this point in time, it takes the wreal input signal, samples it and holds the value until the next sample time.

This simple mechanism has an important draw back that every event on the real input signal between the sampling times is completely ignored. If you simulate the example, you will see that the outcome of the sampler is always 4.0 because it happens to sample the input wave on the 4.0 step. This obviously does not reflect the real behavior very well.

```
module fix_rate_sampler_simple (s_in, s_out );

   input s_in;
   wreal s_in;
   output s_out;
   wreal s_out;
   real   r_out, ts;

   parameter sr = 1M; // sampling rate in Hz

   initial begin
      ts= 1.0E+9/sr; // in ns
   end

   always #ts begin
      r_out = s_in;
   end
   assign s_out = r_out;


endmodule
```

The following sampling block is a bit more advanced. It integrates the input signal over the period between the sampling steps. Thus, all events of the real signal are taken into account. The simulation shows that the output values results in 2.0, which is the time average of the

input signal. The block also considers `wrealXState` and `wrealZState` input values. Whenever one of these values occur the output value is following this assignment.

```
module fix_rate_sampler (s_in, s_out );

   input s_in;
   wreal s_in;
   output s_out;
   wreal s_out;

   parameter sr = 1M; // sampling rate in Hz

   // Input signal averageing based on sampling time
   real T_begin, s_integ;
   real T_last, s_last;
   real ts;
   real avg_out;

   // initialize all values to zero
   initial begin
      s_integ = 0;
      s_last = 0;
      T_begin = 0;
      T_last = 0;
      ts= 1.0E+9/sr; // in ns
   end

   always @(s_in) begin
      if (s_last === `wrealXState)
    begin
       s_integ = `wrealXState;
    end
      else if ((s_last === `wrealZState)
            && (s_integ !== `wrealXState))
    begin
       s_integ = `wrealZState;
    end
      else begin // if s_in is a normal value
       // collecting the value*time integral
       s_integ = s_integ + s_last*($abstime-T_last);
    end
      T_last=$abstime; // save the time of the current event
```

```
    s_last=s_in;     // save the value
  end // always @ (s_in)

  always #ts begin

    // add the last missing time periode to the integral
     if (s_last === `wrealXState)
   begin
      s_integ = `wrealXState;
      avg_out = `wrealXState;
   end
     else if ((s_last === `wrealZState) && (s_integ !== `wrealXState))
   begin
      s_integ = `wrealZState;
      avg_out = `wrealZState;
   end
     else if ((s_integ !== `wrealZState) || (s_integ !== `wrealXState))
   begin // if s_in is a normal value
      // collecting the value*time integral
      s_integ = s_integ + s_last*($abstime-T_last);
      avg_out = s_integ/($abstime-T_begin);  // calc average
   end
     T_last=$abstime;           // same the current time
     s_last=s_in;               // and value
     T_begin = $abstime;        // reset time interval
     s_integ = 0;                // reset integral

  end
  assign s_out = avg_out;


endmodule
```

Slew Limiting

Digital systems mostly react instantaneously on input changes or they might have some delay until the output response is visible. On the contrary, analog systems mostly have a limited rise and fall time, which means when a signal changes from 0.0 V to 3.3 V it will take a certain amount of time for this change. During this period, the signal will – more or less – linearly raise to the final value. This effect is called slew limit. The maximal raise value per time is called slew rate.

Describing this fundamental analog behavior in the discrete domain is not trivial. A single event causes a series of other event to mimic the linear ramp. We need to control the numbers of step that are created for the ramp carefully. Too many events will slow down the simulation unnecessarily while fewer events will be not accurate enough.

The following example is an amplifier that limits the output values and the output slope. The testbench provides a set of different input signals.

```
`include "constants.vams"
`include "disciplines.vams"
`timescale 1ps/1ps
module top();
   wrealAmp_stim stim (P,N);
   wrealAmp      amp (P,N, OUT);

endmodule // top

module wrealAmp_stim (P,N);
output P,N;
wreal P,N;

real Vin;                 // input voltage
real Freq=600M,Phase=0;   // sinusoid params

initial begin  // drive input, comment on expected result:
  Vin=0.1;         // out=1 for DC op point
  #50  Vin=0.108;  // out=1.08 in <20ps
  #100 Vin=0.15;   // out=1.5  in 100ps
  #150 Vin=0.5;    // out sat=3.0  in 300ps
  #400 Vin=0;      // out=0    in 600ps
  #700 Vin=0.2;    // out=2.0 after 400ps, but:
  #300 Vin=0.1;    // prior to full change, ramp down to out=1
  #300 while ($abstime<6000p) begin
                   // generate ramped sine input
    #20 Freq=Freq*1.007;    // gradual freq increase
    Phase=Phase+20p*Freq;   // integrate freq to get phase
    if (Phase>1) Phase=Phase-1;       // wraparound per cycle
    Vin=0.1*(1+sin(`M_TWO_PI*Phase));
                            // sinusoidal waveform shape
  end
  #200 $finish;      // done with test
end
```

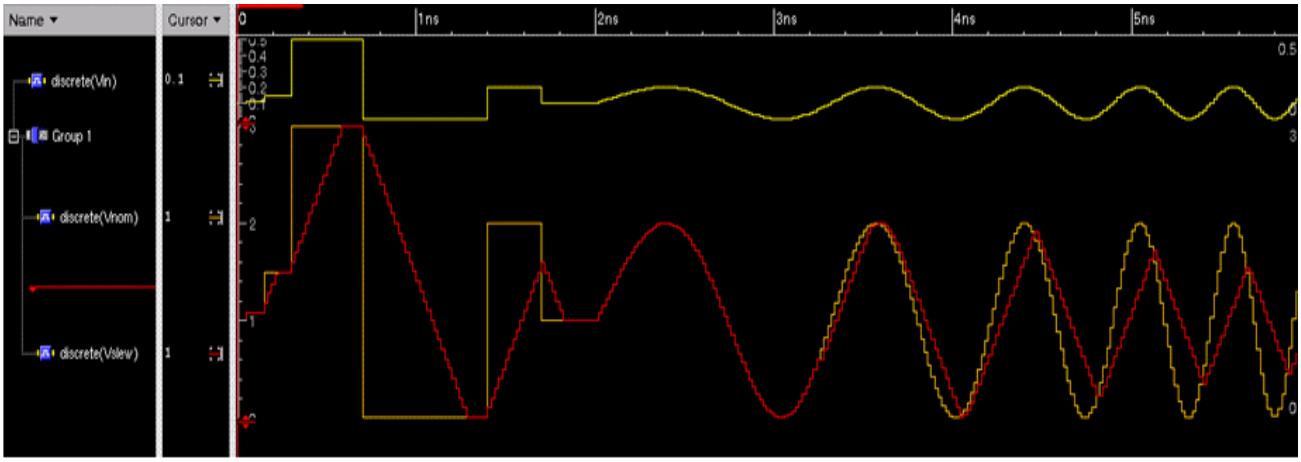
```
assign P = Vin/2;    // drive symmetric diffl signal to inputs
assign N = -Vin/2;

endmodule
```

The amplifier calculates the nominal output value first based on the input and the given limit values. The second always block is responsible to apply the slew limit to the output. The maximum change in value that can be performed depends on the step size since the last event. If the nominal output value change is smaller than this value (`dvlast`) the change can be applied directly to the output.

Larger changes will be limited to the given maximum change value and new events are being created with the given step size of `tstep`. The output is stepping up/down until the nominal output value is reached.

```
// Wreal amplifier with added slew limiting of output signal.
// Input does not require equally spaced timepoints.
// The output waveform will add timepoints at the specified
// tstep spacing when needed to simulate slewing behavior.
`include "constants.vams"
`include "disciplines.vams"
`timescale 1ps/1ps
module wrealAmp (P,N,OUT);
   input P,N; output OUT; wreal P,N,OUT;
   parameter real vio=0, gain=10; // input offset voltage (V),
                                  // gain (V/V)
   parameter real voh=3, vol=0;   // output voltage range (V)
   parameter real slewrate=5G;    // max output slew rate (V/sec)
   parameter real tstep=20p;      // timestep for slew ramp (sec)
   real      Vnom,Vslew;          // nominal output, and slew
   real      dtstep,dvstep;       // time & voltage max step size
   real      tlast=0,dvlast;      // last timepoint, and
                                  // max dV of last step

   always begin                   // compute nominal output value
      Vnom <= min(voh,max(vol,gain*(P-N-vio)));
                                  // linear gain with hard clip
      @(P,N);                     // repeat when input changes
   End

   always begin                   // slew limit the output signal
      if ($abstime==0)
```

```
          dvlast = voh-vol;       // no step limit at DC
      else
          dvlast = slewrate*min($abstime-tlast,tstep);
                                  // max prev step
      tlast = $abstime;           // save event time
      if (abs(Vnom-Vslew) <= dvlast) begin
                                  // If change within +-dV range:
          Vslew = Vnom;           //  step to new value
          @(Vnom);                //  and wait for Vnom to change
      end
      else if (Vnom>Vslew) begin  // Perform slew limit, rising:
          Vslew = Vslew+dvlast;   // limit max change.
          #(tstep/1p);            // and wait for timestep
      end
      else begin                  // Perform slew limit, falling:
          Vslew = Vslew-dvlast;   // limit change per step.
          #(tstep/1p);            // and wait for timestep
      end
   end
   assign OUT = Vslew;            // drive wreal output value
endmodule
```



The figure above shows the input and output waveform of the slew limited filter. It is clearly visible how the red output signal is not able to follow the input signal. It is ramping up or down with linear with the given slew rate.

Modeling Examples

In the following sections, we will discuss a few typical examples of analog models that have been converted to pure wreal models.

Low Pass Filter

The slewing effect discussed above is a typical time domain behavior of analog circuits. Other circuits are more dominated by their frequency domain behavior, similar to filters. Continuous filters, for example, RC, or analog behavioral filters using the `idt/ddt` functions can be implemented directly in the analog domain. However, an equivalent behavior in the discrete domain is not available. A translation for the continuous filter function into a discrete z domain filter is required. This process is called the bilinear transform. References and theoretical details can be found in various textbooks and white papers.

First, let us see the testbench and the analog filter used. Study the code example below.

```
//xrun lpf.vams -input probe.tcl -access r analog.scs -gui
`include "constants.vams"
`include "disciplines.vams"
`timescale 1ns/1ps

module top ();
   electrical Vin, Vout;
   filter i_ctf (Vin, Vout);
   analog begin
      V(Vin) <+ sin($abstime * `M_TWO_PI * 1E8);
   end
endmodule

module filter (Vin, Vout);
   input Vin;
   output Vout;
   electrical Vout;
   electrical Vin;

   real  r_in;
   real  r_in_wreal;
   wreal  r_out;

   parameter real Fp = 1e7;
```

```
   parameter real Dp = 0.5;
   real      Wp;

    initial begin
      Wp = `M_TWO_PI*Fp;
   end
   // electrical implementation
   analog begin
      r_in = V(Vin);
      if (abs(r_in)<100n) r_in=0;
// ignore small oscillations
                // second order low pass filter
      V(Vout) <+ idt(Wp*(
             idt(Wp*(r_in-V(Vout)),2*Dp*(r_in),0,1e-6 ) -
             2*Dp*V(Vout)),
             r_in,0,1e-6);
   end
   // wreal implementation
   always begin
      r_in_wreal = V(Vin);
      #1;
   end
   lpf #(.Fp(Fp), .Dp(Dp), .hfs(1G)) i_lpf_1 ( r_in_wreal, r_out );
endmodule // filter
```

A sine wave is transferred in to the filter module. The core filter is a second order low pass filter implemented by two integration functions. The corner frequency (`Wp`) and the damping factor (`Dp`) are used to parameterize the filter behavior.

The input signal is converted into a wreal signal at a fixed sampling rate. In this example, we know that the input is a sine input signal with a well-defined maximum frequency bandwidth, thus, we do not need to use the complex fix rate sampler block described above. However, for other input signals a more complex sampling routine might be required to avoid aliasing effects. The input sampling rate for the low pass filter implementation is assumed to be a fixed rate. The rate is provided to the filter as `hfs` input parameter. In addition, the sampling rate is a critical factor for real live input signals. The sampling rate limits the frequency bandwidth of the sampled data by the Nyquist criteria, thus, sampling rate of at least twice the maximum frequency of the input signal is required. However, the simulation performance obviously decreases with higher sampling rates. The right sampling rate is a trade off that needs to be made based on the actual characteristics of the input signal and accuracy requirements.

The bilinear transform takes a two-step approach. It converts the analog differential equation into the s-domain and converts the s-domain data into the discrete z-domain in the second step.

The basis differential equation of the filter function described above is shown below:

```
Vout = idt(Wp*(idt(Wp*(Vin-Vout))-2*Dp*Vout))
```

For the transformation into the frequency domain, all differentiations are replaced by a multiplication by `s` while all integrations result in division by `s`. Note that we are neglecting most of the mathematical terminologies and variables naming conventions here to highlight the general process. Refer to standard literature for details.

The above example leads to the following equation:

```
s^2*Vout = (Wp*((Wp*(Vin-Vout))-2*Dp*s*Vout))
s^2*Vout + Wp^2*Vout + Wp*2*Dp*s*Vout = Wp^2*Vin
```

The linear transfer function `H(s)` is calculated as the s-domain output divided by the input:

```
H(s) = Vout/Vin = Wp^2 / ( +Wp^2 + Wp*2*Dp*s + s^2 )
```

In the second step of the transformation, we replace `s` by (g*(1-z^-1)/(1+z^-1)). Where `g` is 2 divided by the sampling rate of the discrete filter.

```
s = (g*(1-z^-1)/(1+z^-1))
H(z) = Wp^2 / ( +Wp^2 + Wp*2*Dp*(g*(1-z^-1)/(1+z^-1))
     + (g*(1-z^-1)/(1+z^-1))^2 )
```

This leads to the discrete transfer function as shown above. The only remaining part of the exercise is to transform the equation in such a way that the filter coefficients for the numerator and dominator are obvious. This is not difficult but still an error prune process, so you need to pay close attention.

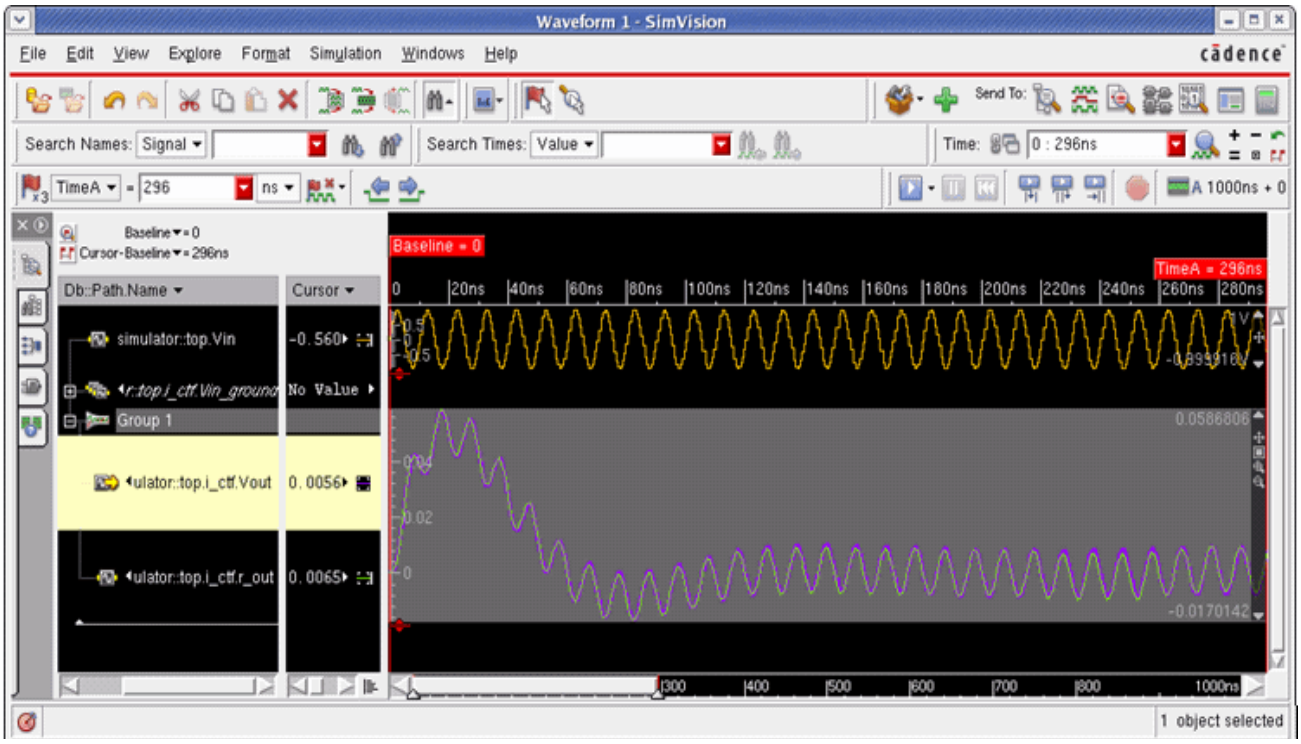
```
H(z) = Wp^2*(1+z^-1)^2 / ( Wp^2*(1+z^-1)^2
        + Wp*2*Dp*g*(1-z^-1)*(1+z^-1)) + (g*(1-z^-1)^2 )
H(z) = Wp^2 + 2*Wp^2*(z^-1) +Wp^2*(z^-2) /
        ( Wp^2 + 2*Wp^2*(z^-1) +Wp^2*(z^-2) + Wp*2*Dp*g
        - Wp*2*Dp*g(z^-2) + g^2 - 2*g^2*(z^-1) + g^2*(z^-2))
H(z) = Wp^2 + 2*Wp^2*(z^-1) +Wp^2*(z^-2) /
        ( (Wp^2 + g^2 + Wp*2*Dp*g) + (2*Wp^2 - 2*g^2)*(z^-1)
        + (Wp^2 - Wp*2*Dp*g + g^2)*(z^-2))
```

After this conversion, we still have the discrete filter coefficients available, so the filter implementation is very easy as shown in the below example.

```
module lpf(Vin, Vout);
   // H(z) = Wp^2 + 2*Wp^2*(z^-1) +Wp^2*(z^-2) /
   //         ( (Wp^2 + g^2 + Wp*2*Dp*g) + (2*Wp^2 - 2*g^2)*(z^-1)
```

```
//        + (Wp^2 - Wp*2*Dp*g + g^2)*(z^-2))

output Vout;
wreal Vout;
input  Vin;
wreal Vin;

parameter real hfs = 1G; // Filter sampling frequency
parameter real Fp = 10M;
parameter real Dp = 0.5;

// LOCAL VARIABLES
real     Ts, g, Wp;
real     num0, num1, num2;
real     den0, den1, den2;
real     yn0, yn1, yn2;
real     xn0, xn1, xn2;

initial begin
   Wp = `M_TWO_PI*Fp;
   yn2 = 0;
   yn1 = 0;
   yn0 = 0;
   xn2 = 0;
   xn1 = 0;
   xn0 = 0;

   // Filter intermediate variables and coefficients calculation
   //
   Ts= 1.0E+9/hfs; // in ns
   g = 2.0*hfs;    // 2/Ts in sec

   // numerator
   num0 = (Wp**2);
   num1 = + 2*Wp**2;
   num2 = +Wp**2;

   // denominator
   den0 = (Wp**2 + g**2 + Wp*2*Dp*g);
   den1 = (2*Wp**2 - 2*g**2);
   den2 = (Wp**2 - Wp*2*Dp*g + g**2);
```

```
   end // always begin

   always @(Vin) begin
      // Signal flow graph intermediate values
      xn2 = xn1;
      xn1 = xn0;
      xn0 = Vin;
      yn2 = yn1;
      yn1 = yn0;
      yn0 = ((num0*xn0) + (num1*xn1) + (num2*xn2)
        - (den1*yn1 + den2*yn2)            )/den0 ;
   end
   assign Vout  = + yn0;
endmodule
```



The figure above shows the results of the low pass filter examples. The output results of the analog and discrete filter implementation are almost identical.

Event-based and Fixed Sampling Time

In the above example, the input signal is first converted into a fixed rate sampled data. The filter block is triggered by these events in the `always @(Vin)` section. This ensures that the filter block is triggered right after the sampling event and a series of filters would trigger one after another in the sequential order.

However, the sampling frequency is a parameter to the filter block itself that is used to calculate the filter coefficients. A mismatch between the actual sampling rate and the filter parameter setting would result in wrong output results. A measurement of the sampling rate inside the filter replacing the instance parameter might be a useful enhancement to overcome this limitation.

The other task would be to change the always block triggering to a time-based trigger, such as `always #Ts begin`. This would ensure that the filter input is sampled at the right rate and that it is not driven accidentally by a flexible sampling rate. However, the comments above about choosing the right sampling rate still apply.

Furthermore, if a chain of filters triggers at the same time point then the sequence of execution of these filters is unpredictable. Thus, it might be that the sequence of blocks is executed in the wrong order.

Generally, event-based modeling and modeling with fixed sample times are both adequate measures in real value modeling. However, we need to choose the appropriate technique while considering different cases.

ADC/DAC Example

The next example is an ADC and a DAC combination. The testbench is creating different kind of input stimuli including a ramp and a sinusoidal waveform similar to the value sources discussed earlier. In addition, VDD, VSS, and a CLK value are created.

```
`timescale 1ns/1ps
`define Nbits 12
`include "constants.vams"
`include "disciplines.vams"

module top ();

   wreal AIN, AOUT, VDD, VSS;
   wire [`Nbits-1:0] DOUT;
   wrealADC I_ACD (DOUT,AIN,CK,VDD,VSS);
   wrealDAC I_DAC (AOUT,DOUT,CK,VDD,VSS);
```

```
   real r_ain, r_vdd, r_vss;                  // input voltage
   real Freq=600K,Phase=0;   // sinusoid params
   reg clk;

   initial begin
      clk = 0;
      r_vdd = 3.3;
      r_vss = 0.0;
      r_ain=0.1;          // out=0.1 for DC op point
      repeat (10) #10 r_ain=r_ain+0.348;  // increasing ramp
      repeat (10) #10 r_ain=r_ain-0.339;   // falling ramp
      #30 while ($abstime<6000p) begin
                              // generate ramped sine input
     #2 Freq=Freq*1.0007;        // gradual freq increase
     Phase=Phase+2n*Freq;        // integrate freq to get phase
     if (Phase>1) Phase=Phase-1; // wraparound per cycle
     r_ain=1.8*(1+sin(`M_TWO_PI*Phase));
         // sinusoidal waveform shape
      end
      #200 $finish;                 // done with test
   end

   always #2 clk = ~clk;

   assign AIN = r_ain;
   assign CK  = clk;
   assign VDD = r_vdd;
   assign VSS = r_vss;

endmodule
```

The wreal input signals are passed to an ADC block. The first always block calculates the lower and upper limits for the output values and the input value precision (`PerBit`) given by the input value swing and the number of output bits. These values are updated, whenever the supply values are changing.

The second always blocks is triggered on every change of the clock. If the input value does not exceed the lower or upper limits, the value is divided by the `PerBit` value providing the output value in the integer format. This value is assigned to the output bus after a given delay.

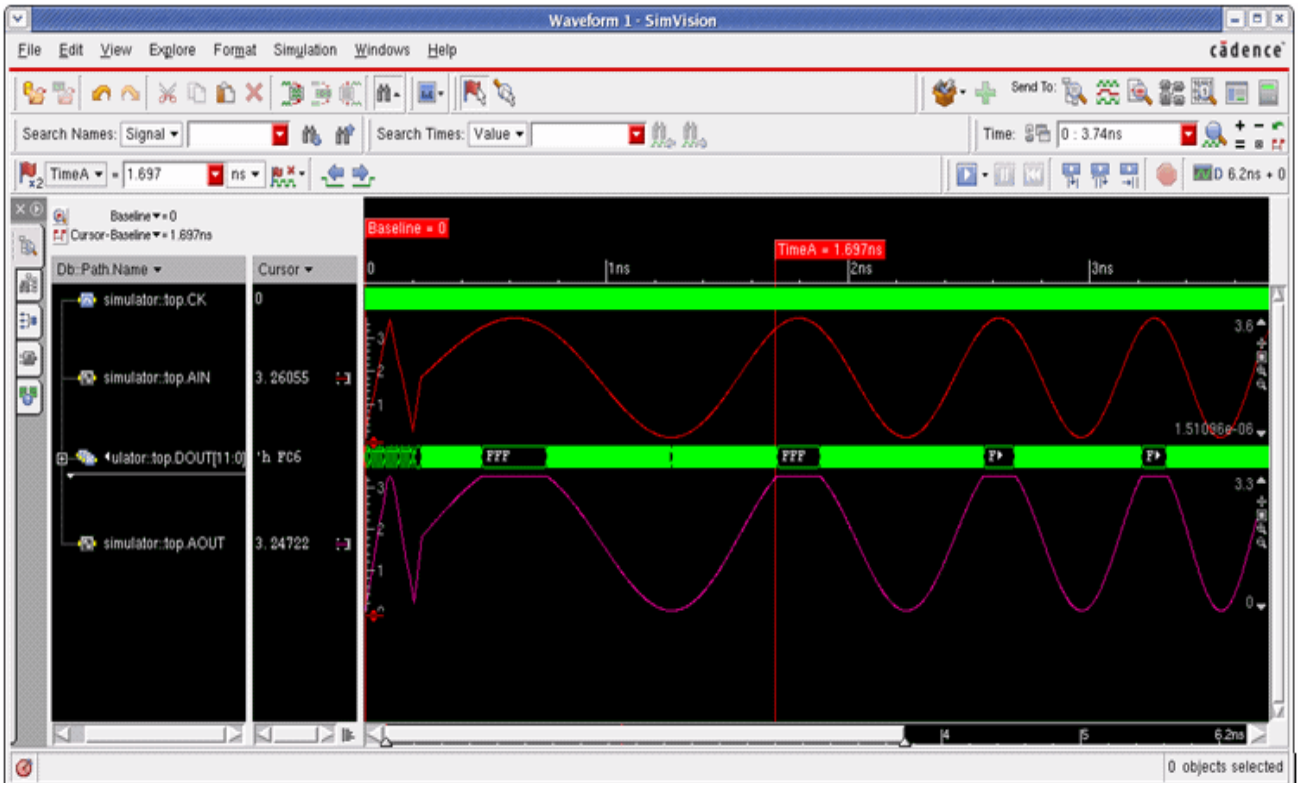
```
module wrealADC (DOUT,AIN,CK,VDD,VSS);
   output [`Nbits-1:0] DOUT;
```

```
   input        CK;
   input        AIN,VDD,VSS;
   wreal        AIN,VDD,VSS;
   parameter   Td=1n;
   real        PerBit, VL, VH;
   integer     Dval;
   always begin     // get dV per bit wrt supply
      PerBit = (VDD-VSS) / ((1<<`Nbits)-1);
      VL = VSS;
      VH = VDD;
      @(VDD,VSS);// update if supply changes
   end
   always @(CK) begin
      if (AIN<VL) Dval = 'b0;
      else if (AIN>VH) Dval = {`Nbits{1'b1}};
      else Dval = (AIN-VSS)/PerBit;
   end
   assign #(Td/1n) DOUT = Dval;
endmodule // wrealADC
```

The inverse digital to analog (or wreal in this case) operation is even simpler. The first always block is almost identical and the second just multiplies the output precision value with the input value bus. The resulting value is assigned to the wreal output wire.

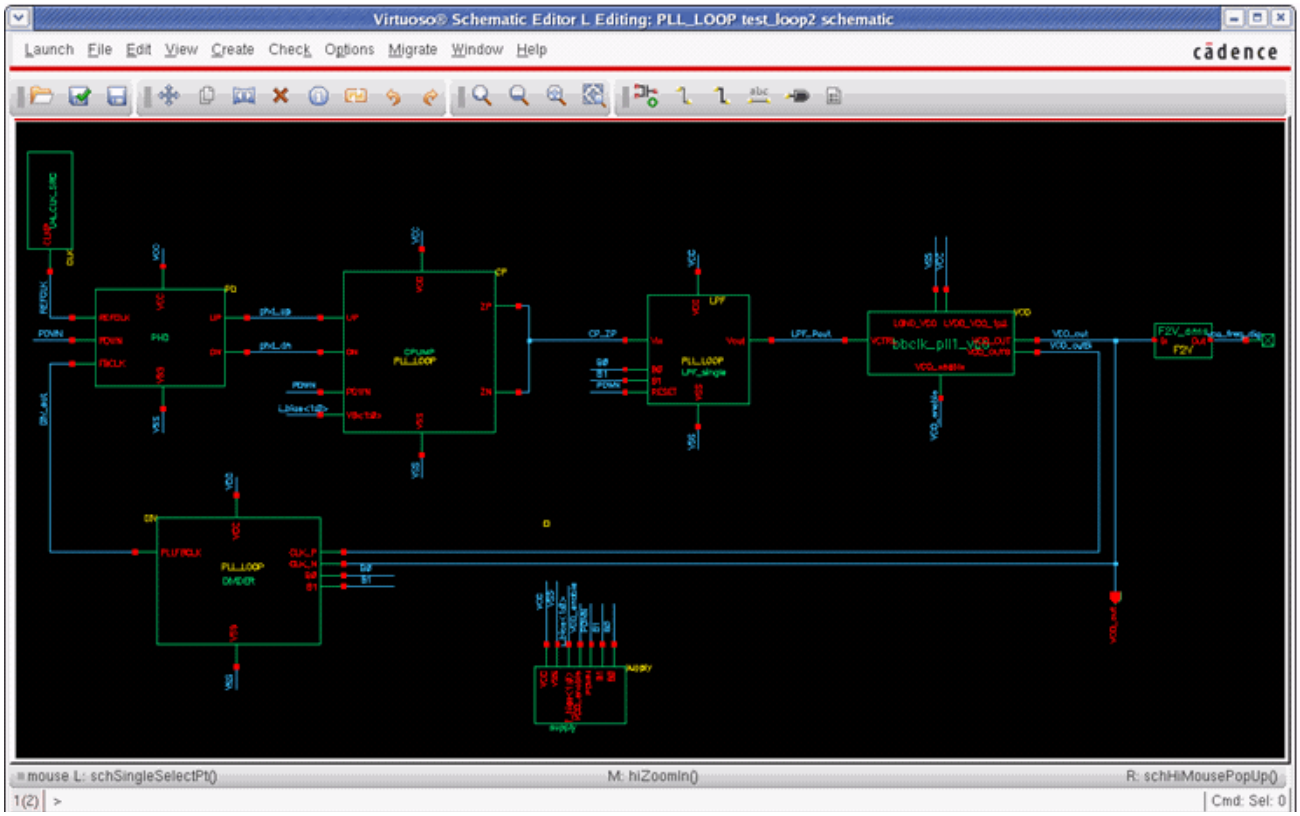
```
module wrealDAC (AOUT,DIN,CK,VDD,VSS);
   input [`Nbits-1:0] DIN;
   input        CK,VDD,VSS;
   output       AOUT;
   wreal          AOUT,VDD,VSS;
   parameter real Td=1n;
   real       PerBit,Aval;
   always begin // get dV per bit wrt supply
      PerBit = (VDD-VSS) / ((1<<`Nbits)-1);
      @(VDD,VSS);// update if supply changes
   end
   always @(CK)  Aval <= VSS + PerBit*DIN;
   assign  #(Td/1n) AOUT = Aval;
endmodule // wrealDAC
```

The following figure shows the simulation results for the above testcase.



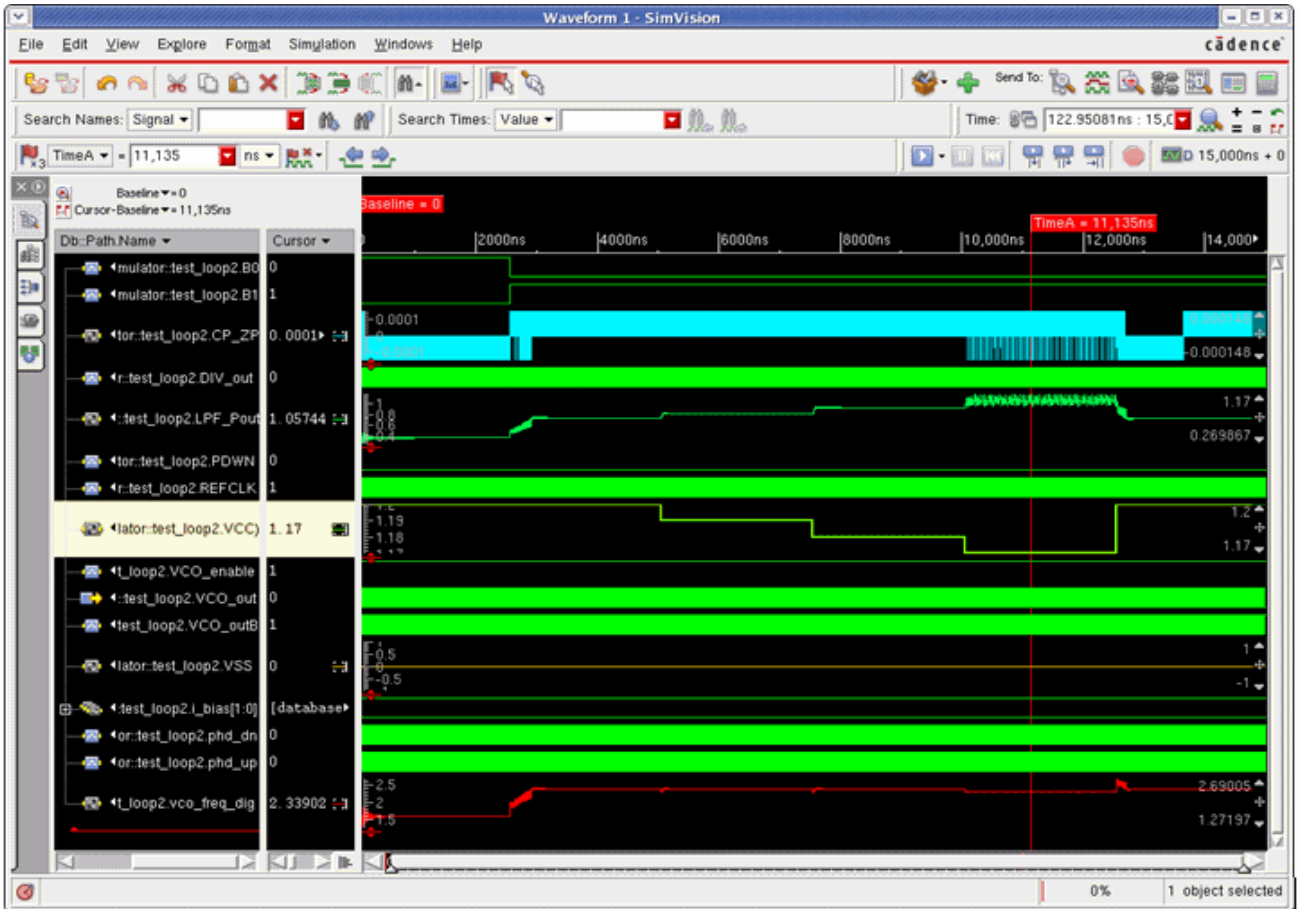
Case Study

We are using a phase locked loop (PLL) example to demonstrate the wreal modeling approach in a more complex scenario. The PLL was designed in the Virtuoso Schematic environment. The top-level schematic is displayed below:



Let us now see the functions of all the blocks in the PLL. The upper left block is the reference clock generator, example, a crystal oscillator. The next block, PHD, is the phase detector that compares the phase of the oscillator with the phase of the signal inside the PLL loop. Depending on which signal is first in terms of phase it creates a signal on one or the other output pin. The next block is the charge pump (CP). A charge pump translates the PHD output signals in a small amount of charge that is pumped into or pulled out of the following low pass filter (LPF). In a perfectly locked PLL, the output of the charge pump would have a constant voltage. If the frequency of the PLL is too small according to the reference, the output voltage is increased a bit. If it is too high, the voltage level is decreased. The LPF block is filtering out high frequency portions of the signal to stabilize the loop. Finally, the voltage-controlled oscillator (VCO) is generating the output signal. This signal might have a higher frequency, example, three times higher, as the input reference. To close the loop, the VCO output frequency is divided by this factor, 3 in the example. Now the reference signal and the downscaled output frequency can be compared in the PHD as described above.

The two additional blocks on the schematic is a frequency measurement block on the extreme right hand side and the source of all the power and reference signals at the bottom.



The PLL starts in a divide by 2 mode resulting in a 1.6 GHz output frequency given the 800 MHz reference clock. At about 2.5us the mode it switches to a divide by 3, thus, the output frequency stabilizes at 2.4 GHz after some time. Now the supply voltage drops in three steps from 1.2 V down to 1.17 V where the VCO cannot deliver the 2.4 GHz anymore and the PLL goes out of lock. It recovers after the VCC voltage rose again.

Let's see some specific wreal modeling features used in this example.

The supply block used is described below:

```
`include "disciplines.vams"
`timescale 100ps / 10ps
module supply ( B0, B1, PDWN, VCC, VCO_enable, VSS, i_bias );

   wreal  i_bias[1:0];
   wreal VCC;
```

```
   wreal VSS;

   parameter real pvcc = 1.2;

   real      v_supply;

   assign    i_bias[1] = 50u;
   assign    i_bias[0] = 50u;
   assign    VCC = v_supply;

   initial begin
      // initial settings
      v_supply = pvcc;
      #25000 // after another 2.5 us reduce v_supply
    v_supply = v_supply - 0.01;
      #25000 // after another 2.5 us reduce v_supply
    v_supply = v_supply - 0.01;
      #25000 // after another 2.5 us reduce v_supply
    v_supply = v_supply - 0.01;
      #25000 // after another 2.5 us recover VCC
    v_supply = pvcc;
   end
endmodule
```

Let us now focus on the power supply pin VCC. It is defined as an output of type wreal. Internally, we use a real variable `v_supply` to assign the different voltage levels at specific time points – as we would do in pure digital Verilog. We assign the `v_supply` real variable to the real-wire VCC. Note the concept of event-based changes of the values at specific time points (discrete time).

Another important observation is that the wreal values have no unit. While we intuitively think of VCC being a voltage level – which is in reality – but it is not specified explicitly. The wreal port just carries a real value without a unit.

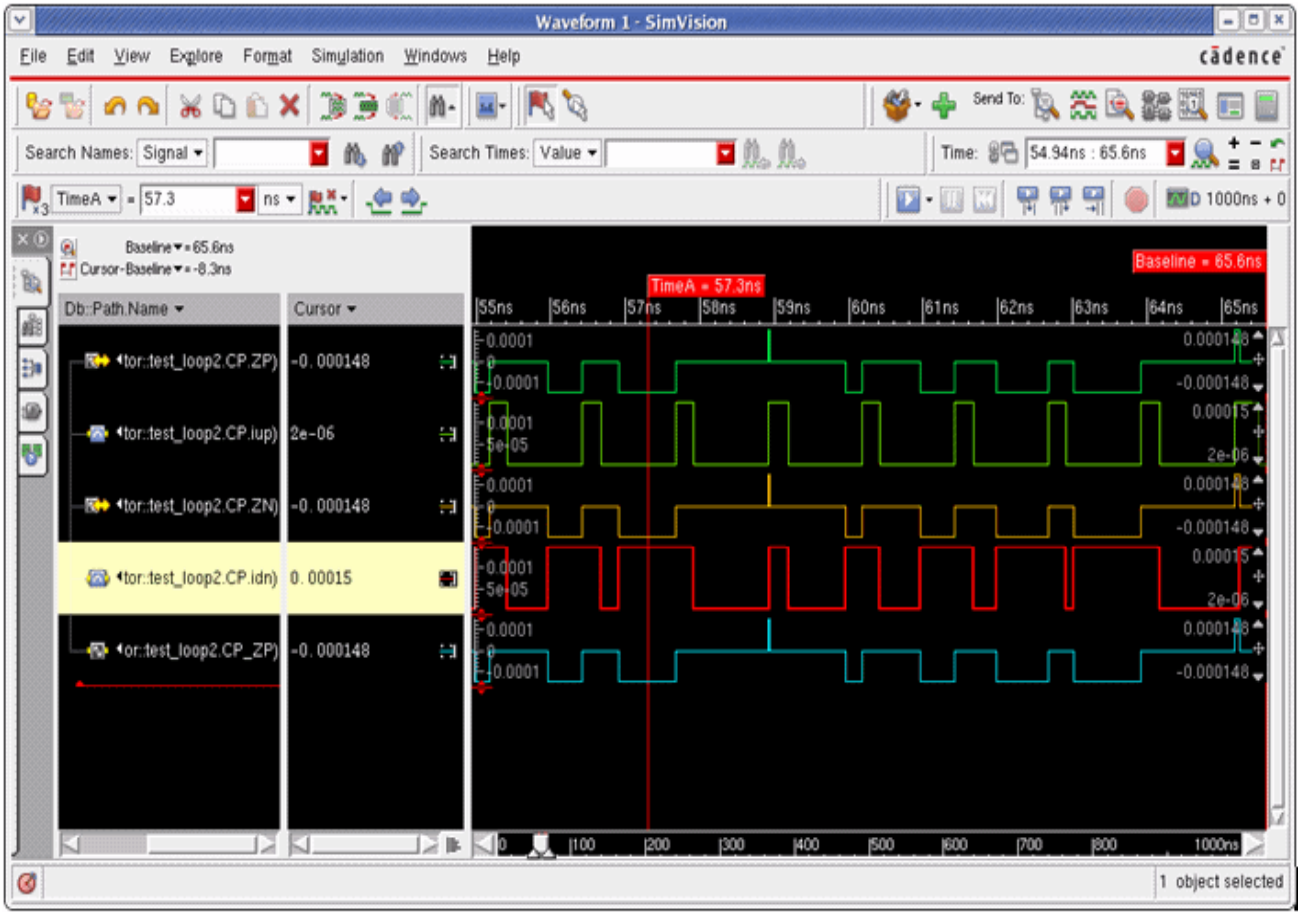
Wreal also supports arrays as shown above. In this case, two bias current values for the charge pump are defined in the supply module. The use of arrays follows the Verilog concept and syntax.

Let us review the use of X and Z states in this example. The concept of an unknown – X and high impedance – Z state that is used in the 4-state logic is useful for wreal signals as well. The meaning of X and Z is equivalent for the wreal case. The charge pump example uses the X value in case the enable signal is low:

```
always @(ENABLE,upReal,dnReal,I_out) begin
    // calculate up or dn current values based
    // on the validity of supply
    // and enable signals
    if (ENABLE===1'b1) begin
       iup = (upReal>0) ? IUPscale*I_out : 2u;
       idn = (dnReal>0) ? IDNscale*I_out : 2u;
    end
    else begin
       iup = `wrealXState;
       idn = `wrealXState;
    end
 end
```

See the schematic figure shown above. Find the outputs of the charge pump (Zp, Zn) connecting to the single filter input. Diving into the charge pump source code, the result was that both the output values are actively driven:

```
assign ZP = iup;
assign ZN = -idn;
```



In the time point shown above, iup drives the net CP_ZP to a value of 2 u while idn drives the same net to 0.15 m. This would result in a conflict since two drivers are driving the net at the same time toward different values.

The introduced concept of wreal resolution functions resolves this limitation. In the simulation, we used the `-wreal_resolution sum` switch to `xrun` to resolve the connected net to the sum of both values. In the original design, this CP_ZP node was a current summing node connecting the two charge pump outputs. To mimic this behavior in wreal, the above resolution function sum is used.

Another useful enhancement is the `$table_model` function known from analog Verilog-A and Verilog-AMS blocks. The function is now available for real values as well. The table model function enables an easy calibration of a real value behavioral model against measured data provided as a text file. The VCO in this example uses this capability.

```
real input_value;
   always @(LVDD_VCO_1p2, LGND_VCO, VCTRL) begin
```

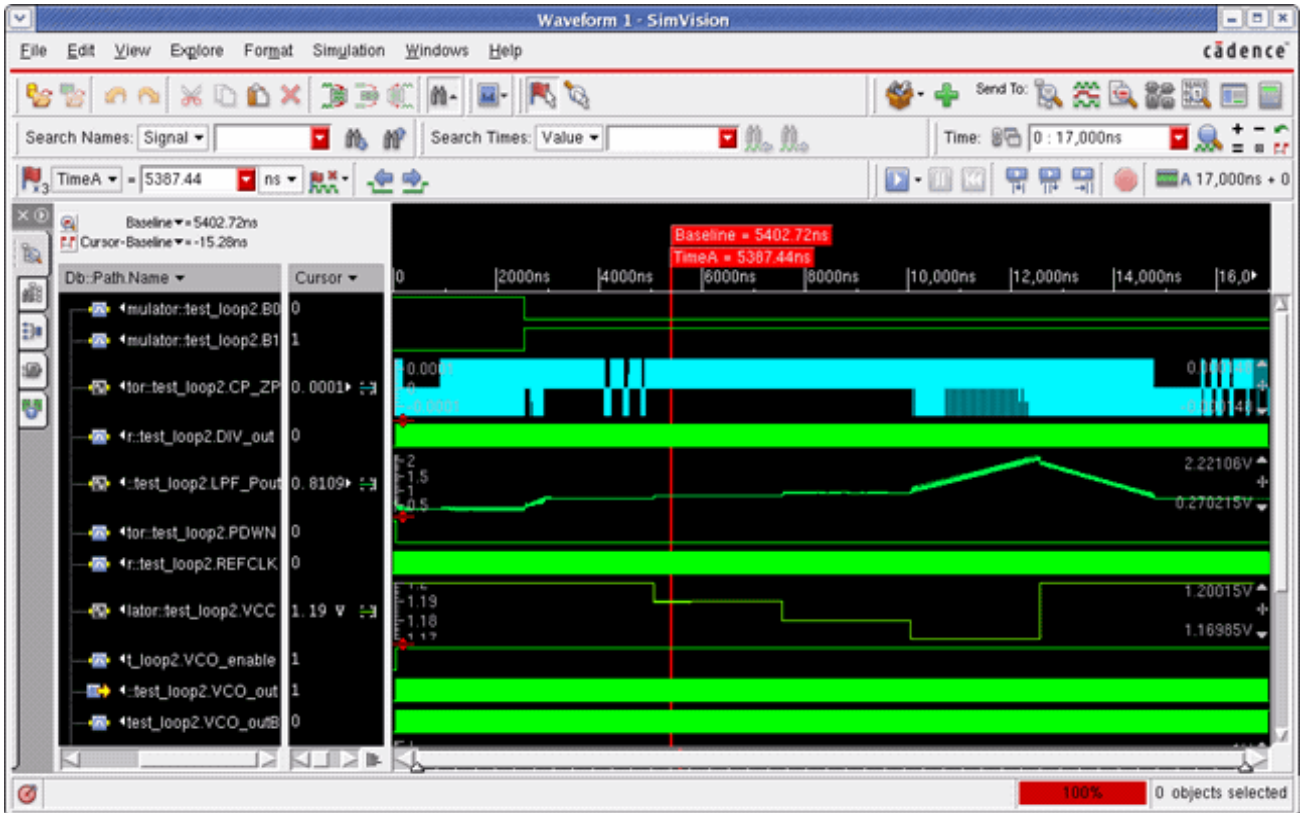
```
      input_value = (LVDD_VCO_1p2 - LGND_VCO);
      tableInstantaneousFreq = $table_model(input_value, VCTRL,
                        "vtuneFreqControl.tbl","1CC,3CC");
   end
```

The example above calculates the VCO output frequency according to the table in the text file `vtuneFreqControl.tbl`. The text file contains the following data that is collected during the characterization of the original VCO.

```
#VDD    VCNTL   Freq
1       0       9.81E+06
1       0.4     1.05E+09
1       0.8     1.41E+09
1.1     0       9.88E+06
1.1     0.45    1.29E+09
1.1     0.9     1.52E+09
1.2     0       9.95E+06
1.2     0.3     1.37E+09
1.2     1       2.69E+09
...
```

Similar to the mechanism of automatically inserted connect modules (AICM) between the continuous and discrete domains (electrical/logic) are connect modules available between wreal and electrical. These are called E2R and R2E. They are also inserted automatically if needed during the elaboration process. That ensures an easy replacement of blocks from

different types of representations. In this case, we chanced the filter into an electrical filter to see results.

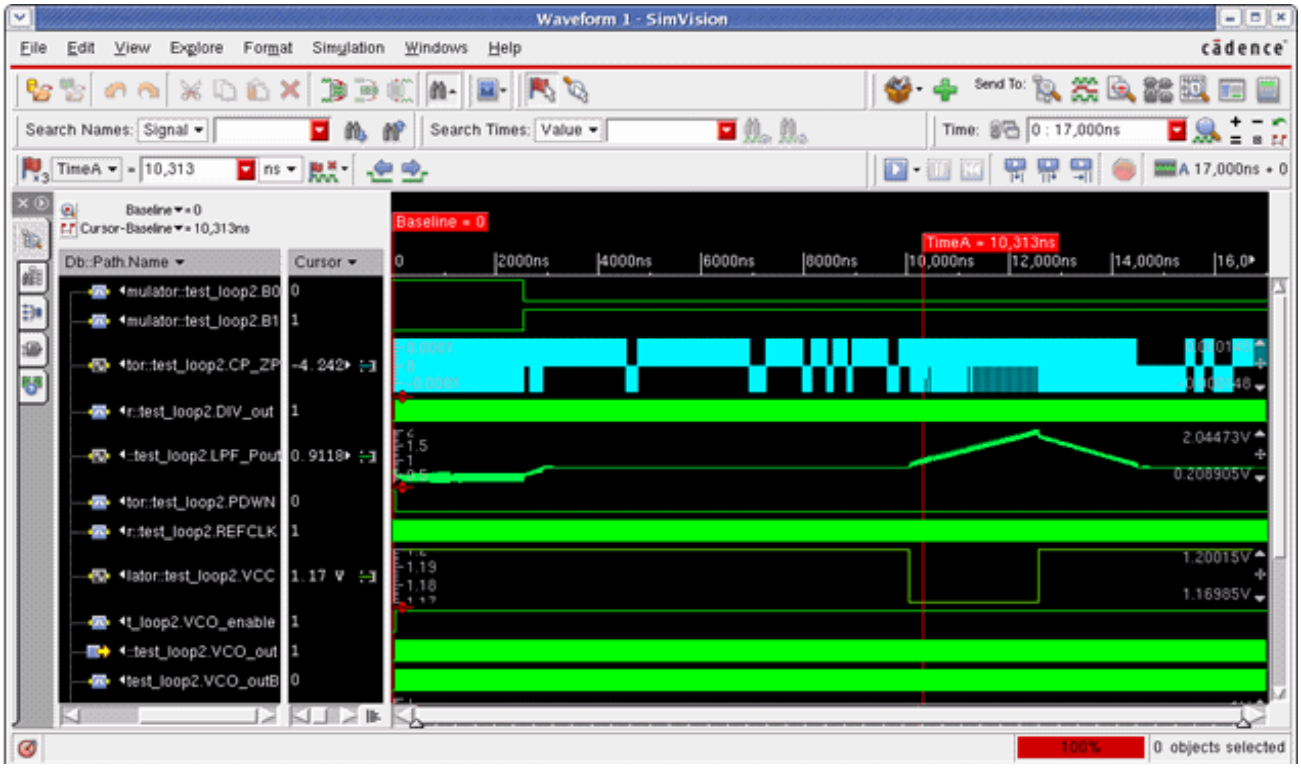


You might have noticed that the `LPF_Pout` value rises higher than the supply voltage in this case (around 12 us). This is a modeling artifact because we used an ideal charge pump with a wreal signal flow output (not feedback) that creates current event though the voltage level is above the supply voltage range. In reality, the charge pump current would settle around the supply voltage level.

The IE Report message in the `xrun` logfile (use the –`iereport` option).

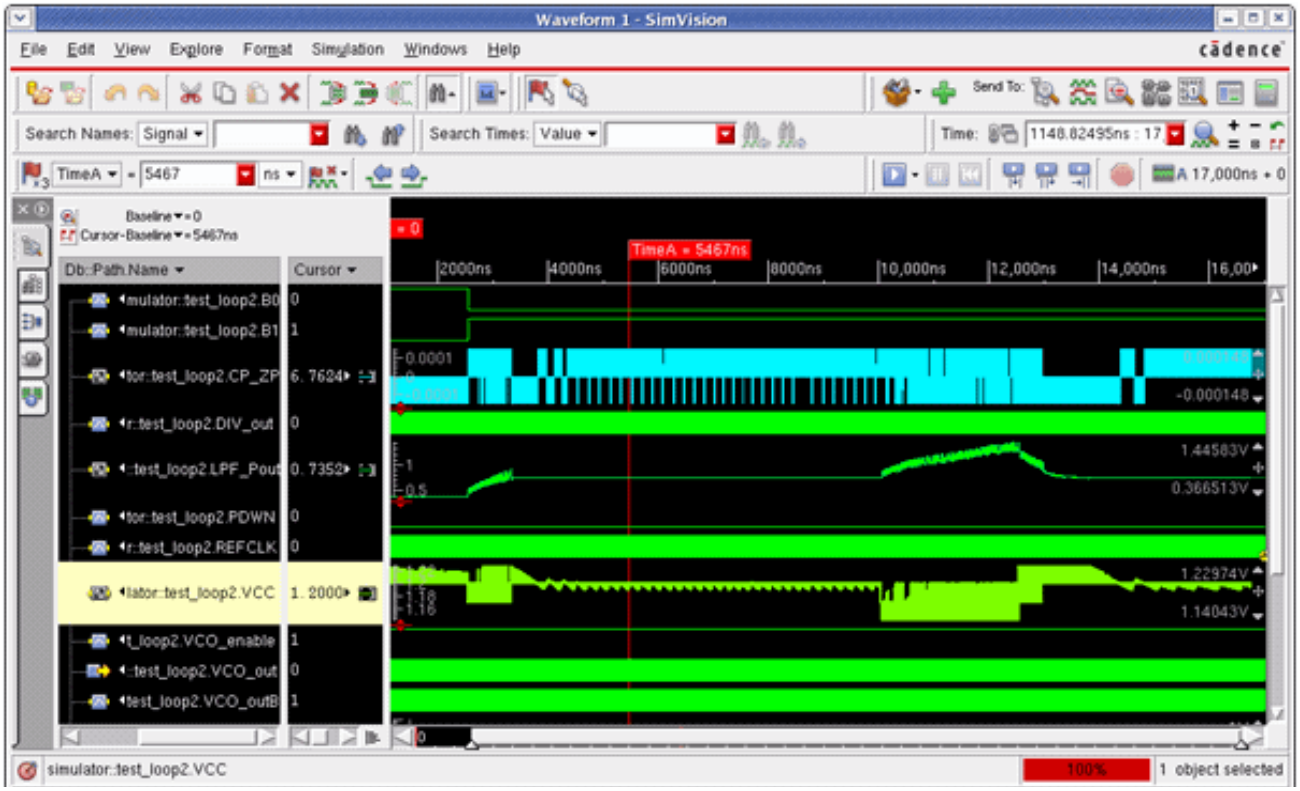
```
IE Report Summary:
  E2R ( electrical input; wreal inout; )          total: 1
  ER_bidir ( wreal inout; electrical inout; )     total: 2
------------------------------------------------------------
Total Number of Connect Modules                 total: 3
```

The proper settings of the CM parameters are critical for correct simulation results, as we will see in the next exercise. If we change the `vdelta` parameter back to it's default value of `` `Vsup/64:(... #(.vdelta(`Vsup/64) ...) `` the simulation results look quite different.



Vsub/64 results in about 0.028 V for the given supply of 1.2 V, thus, only changes larger than this value are creating events on the wreal net. Consequently, the first two supply voltage drops of 0.01 are not seen on the top-level VCC net.

Consider another scenario where we change the internal resistance of the R2E_2 and R2E_bidir CM back to their default value of 200 Ohm. What will happen now?



The electrical filter is consuming some amount of current from the VCC power supply that is generated from the wreal source. The power net is now connected by a 200 Ohm resistor to the wreal supply. These results in a significant IR drop on the VCC line and consequently in a changing VCC supply level. The simulation results are far from being correct anymore. As you saw in the last few simulations, the parameter settings of the wreal CM are critical and need to be set carefully to achieve good results.

5

Conclusion

This book introduced some of the Real Valued Modeling technologies provided in the IES 8.2 release, and an example of how they can be applied to a real-world modeling problem.

The full-chip SoC simulations now require much more accuracy than what digital Verilog/ VHDL alone can offer. Real value models are being used more often to discretely model the analog portions of the design. The real value models are easily portable between design and verification environments. Real models work in Virtuoso® and Xcelium™ environments and can be used as the hand-off between analog and digital designers. Overall, this methodology can provide you significant simulation speed improvements, while enabling you to perform top-level verification of your mixed-signal SoCs.

6

Appendix A: Advanced Digital Verification Methodology

This appendix focuses on a generic outline of the digital verification flow in order to relate the important intersections with analog and to the challenges faced when bringing the two together for mixed-signal verification.

A simplified view of the digital verification flow as it applies to implementation is as listed below:

- Functional verification through RTL simulation.
- Equivalence checking of RTL and gate level.
- Static timing analysis throughout the design.

The most costly as well as critical verification step at relatively high level of abstraction is the functional verification. Equivalence checking provides the verification against the synthesized gate design as well as other checks for structural, functional, and electrical integrity (refer to Conformal/CLP documentation for details). The timing of the final place and route implemented physical design is checked by the static timing analysis step.

Equivalence checking generates mathematical models of the two different circuit representations in order to determine the equivalence. There are some theoretical approaches to apply those techniques to analog design as well, but in general, this step is not related to mixed-signal verification.

Since static timing analysis is only applicable in context with a reference clock, it is not directly useful for analog blocks. However, it is important to consider analog parts when integrating the analog IP into a SoC, since the analog block might be part of a critical path through the design. This situation is solved either by running detailed performance simulations using mixed-signal tools to check the timing or by creating a timing model for the analog blocks and using this model during standard static timing analysis. Although there are characterization tools that build timing models for standard digital blocks, currently no standard tools will automatically build the regression suites and take the measurements needed for a timing model on analog IP. The method employed here is leveraging the analog design environment

setting up automatic regression suite for each analog architecture and then having the analog environment generate the input data for the timing model.

Not long ago engineers wrote directed test cases, which were composed of simple sequences of ‘0’s’ and ‘1’s’ as input stimulus for the design to do functional verification. Checking was done manually: it was part of the test and verification engineer’s task to explicitly predict the expected response of the DUT for a specific test. In some cases, it required even visual inspection of a waveform. Since this extra work had to be repeated for each and every testcase whenever there was a design change, engineers tended to minimize the amount of hard coded checks within their tests, increasing the risk of a bug escape. Configuration and reuse of those test benches were very limited. To track the verification progress, huge text documents or tables were used to handle the test implementation and results.

One of the biggest limitations of this method is the directed test approach. The designer is trying to test one specific feature of the specification in a dedicated test. Most likely, this test will pass without problems since the feature was implemented with the same set of implicit assumptions as the test. However, most errors occur when a condition is triggered in a non-standard way. This is the main reason why modern verification approaches are based on constraint random tests.

The challenge of digital functional verification comes down to answering the question of whether or not enough simulations have been run and all corner cases have been covered. Only additional measures, such as code coverage, assertion coverage, and functional coverage provide this information and enable an easy assessment of the current verification status. Finally, reuse and automation are much better supported in modern verification techniques (see below).

Verification Plan and Metric-driven Verification

The process of creating a verification plan is initiated with a verification planning session. All parties involved in the design project should participate in the planning session to ensure that the plan does contain all needed information and the verification goals are agreed upon between the team members. The document describes what needs to be verified in a hierarchical way.

In most cases, the verification plan is a living document, which means the plan is refined, modified, enhanced during the design and verification process. This is a natural process since the design, spec, and experiences are changing during the design phase. Clearly defined verification goals are very helpful for the whole verification process, even if the goals are not formally written down in a verification plan.

Advanced verification techniques have been developed and introduced into today's digital design flows to overcome the limitations and productivity restrictions. Moreover, the prediction of verification quality is a major improvement in the state of the art verification methods. Main components of these verification techniques are:

- Automated random stimulus generation
- Automated self-checking (assertions and reference models)
- Coverage measurements and tracking
- Adding formal methods into the verification flow

The design under test (DUT) is stimulated with some input data, simulated, and the simulation results are stored. There are two important tasks:

- Generation of input stimuli
- Checking the results against expectations

A third task is functional coverage to measure which verification goals have been achieved during simulation.

Metric-driven Verification and Advanced Testbench

A metric-driven verification flow assembles the pieces together that have been discussed above. The simulation results are automatically checked and problems are being reported. An automatic stimuli generator creates tests on a random basis within given constraints. On top, the designer might have a certain amount of tests that are pre-defined and need to be run (directed tests) to reach certain corner cases. All of this is enabled by some advanced testbench that takes care of the different simulation scenarios. The testbench is typically written in e or SystemVerilog, whereas others use SystemC, Verilog, or VHDL as well.

7

Appendix B: Analog Simulation Today

Analog verification is based on the idea of simulating the circuit with a given input stimulus and observing the correct output behavior. In most cases, this process is still based on manual waveform inspections. There are several reasons for this:

- Necessary measurements and calculations on analog waveforms are sometimes hard to formulate in a mathematical way while manual check is very easy and fast for an experienced designer.
- Waveform inspection is a major part of the analog workflow. People are simply used to eyeballing the results.
- Analog design is not very formalized and still relies heavily on expert knowledge. This implies that there are significant amount of implicit assumptions beside the specification that need to be taken into account.

Even though a complete replacement of manual waveform inspection in analog design may not be realistic today, it is relatively easy to automate the straightforward checks in the analog working environment.

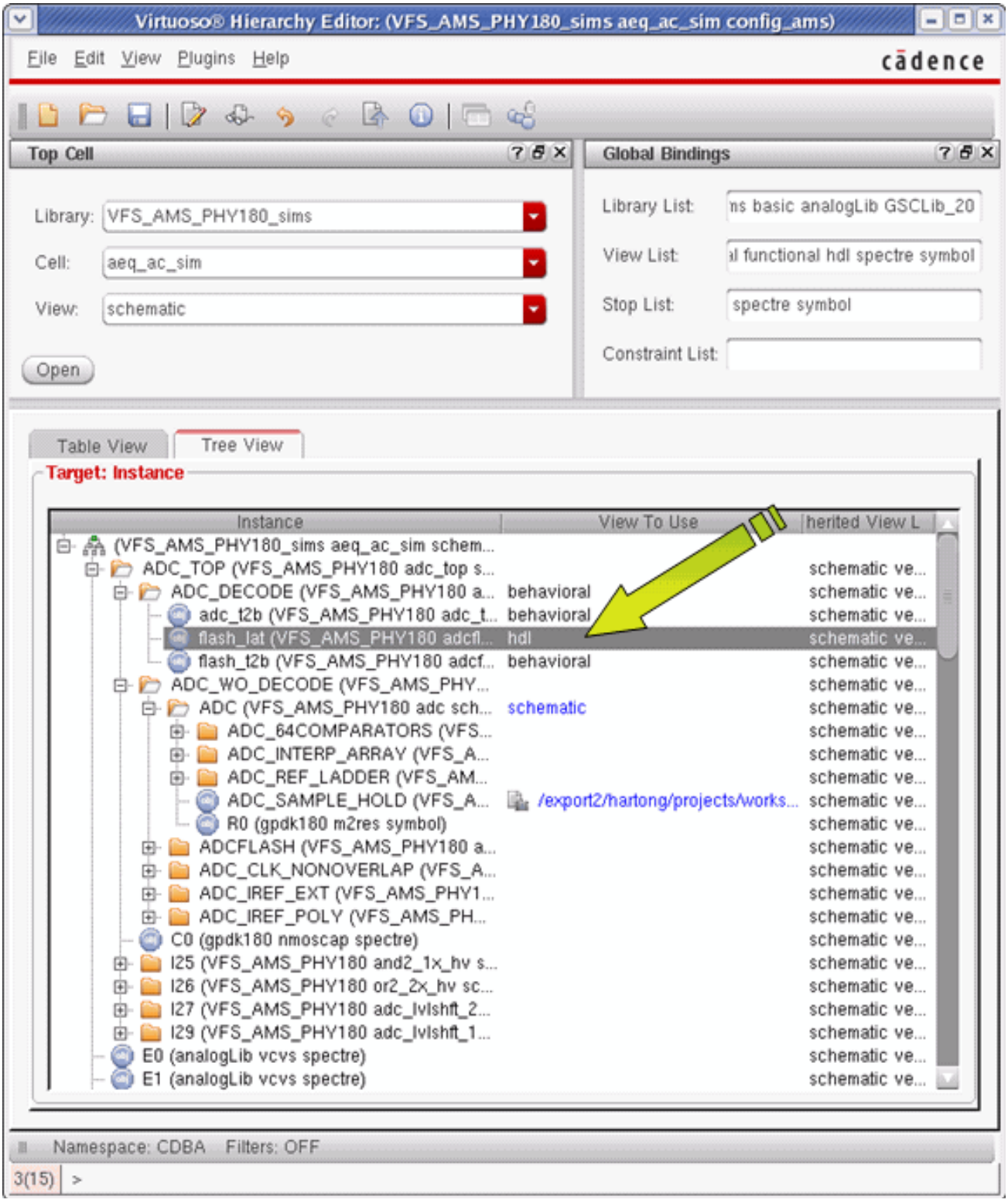
For example, the Virtuoso Analog Design Environment (ADE) is a very common workplace for analog designs. It provides an analog-centric verification suite using the following features:

- Support for transient, DC, AC, and RF analysis
- Specification-driven design
- Supports multiple tests/analysis and measurement inside a single state
- Parasitic aware design flow
- Constraint-driven design
- Automatic characterization and model generation
- Integrated sizing and optimization capabilities
- Report generation
- History archive

The ability to automate the verification process combining a complete set of test benches, tests, and analyses into a single execution task is a huge productivity benefit for the analog designer. Combining this with a complete set of measurements checking the specification values, results in a simple but effective pass/fail summary for each specification value. A rich set of predefined checking functions and a straightforward scripting language allow users to implement circuit specific checks in a very efficient way. Moreover, the scripting language provides an easy and flexible way of automating the analog verification in the batch mode.

In analog design, device parameter variations due to the manufacturing process and changes in temperature, aging effects need to be taken into account. This is mainly ensured by sweeping parameter in a set of verification runs or by statistical methods, such as Monte Carlo and Corner analysis. These types of analyses are supported and used in ADE for many years.

The design data in the analog world is mostly organized in the design framework II (dfII) database. The blocks are categorized in libraries, cells, and views. Each cell in the design may have different representations – views. A graphical tool, namely Hierarchy Editor (HED), is used to bind each cell in the design to the appropriate view. The resulting configuration is saved as a cellview.



There are varieties of different analog solvers available. SPICE-level solvers, such as Spectre, Spectre RF, Spectre Turbo, and APS, are designed to solve the SPICE-level equations with the full analog accuracy. The fast SPICE engine, Ultrasim, on the contrary can trade off accuracy versus speed. Thus, the user can chose to reduce the accuracy level for some blocks, example, digital blocks, in order to gain simulation performance and capacity.

The simulation approaches in analog and digital domains are fundamentally different due to the structure of the underlying equation system to solve. While the digital solver is solving logical expressions in a sequential manner based on triggering events, the analog simulator must solve the entire analog system matrix at every simulation step. Each element in the analog design can have an instantaneous influence on any other element in the matrix and vice versa. Thus, there is no obvious signal flow in a particular direction. Time and values are continuous. In digital, time and values are discrete. The solver can apply a well-defined scheme of signal flow and events to solve the system.

This results in different run time complexities for the analog and digital solver. The digital event-based engines have an almost linear run-time complexity in terms of design elements. For example, a design that is twice as large as a reference would approximately consume the double simulation resources. In the contrary, the analog solver need to setup and invert the system matrix entirely. The matrix inversion has a run-time complexity slightly better than quadratic (depending on the algorithm used). Thus, an analog design of a double size needs more than four-time more simulation resources to be solved. This results in hard limitations for feasible circuit sizes for analog simulation. Tool enhancements, such as fast SPICE engines (Ultrasim) and parallel solving of the system matrix (APS) push these limits further away, however, the general trend remains.

Mixed-signal Simulators

Commercial mixed-signal simulation solutions – like AMS Designer – are available since early nineties and the tools progressed over the last decade to powerful engines managing multiple power supply domains, bidirectional interface connections, varieties of analog solving algorithms. Mixed-signal extensions to the standard behavioral languages (Verilog-AMS and VHDL-AMS) provide flexible modeling capabilities. Given that, mixed-signal simulation in itself can be considered as a solved problem.

However, simulation is only the enabling part for the verification process. The verification environments are still separated in an analog driven flow or a digital-centric methodology as described above. The choice of the right level of design abstraction is especially important when moving to mixed-signal simulation.

Mixed-signal simulation is used mainly by the analog design team. Consequently, the use model is aligned with the analog workflow. AMS Designer is integrated in the ADE use model. It can read design information from the dfII database as well as file-based descriptions.

A significant demand for mixed-signal simulation within the digital use model has been visible over the last couple of years. AMS Designer's command line use model fulfills these requirements. A simulation can be setup and started easily from the command line. This enables a straightforward integration into the digital-centric verification flow as well as all the flexibility needed for the mixed-signal simulation.

The analog content is in many design flows is represented only as Verilog-AMS or SPICE-level netlists. The goal was to make the flow as simple as possible, however, compared to a pure digital simulation, some additional information also need to be provided.

- Settings for the analog solver
- Including analog (SPICE) content
- Configure the design parts that should be replaced by analog blocks
- Define the Connect Modules (CM) to connect analog and digital signals
- Define the port mapping information between SPICE and Verilog