

Analog Behavioral Modeling with Verilog-A

Boris Troyanovsky

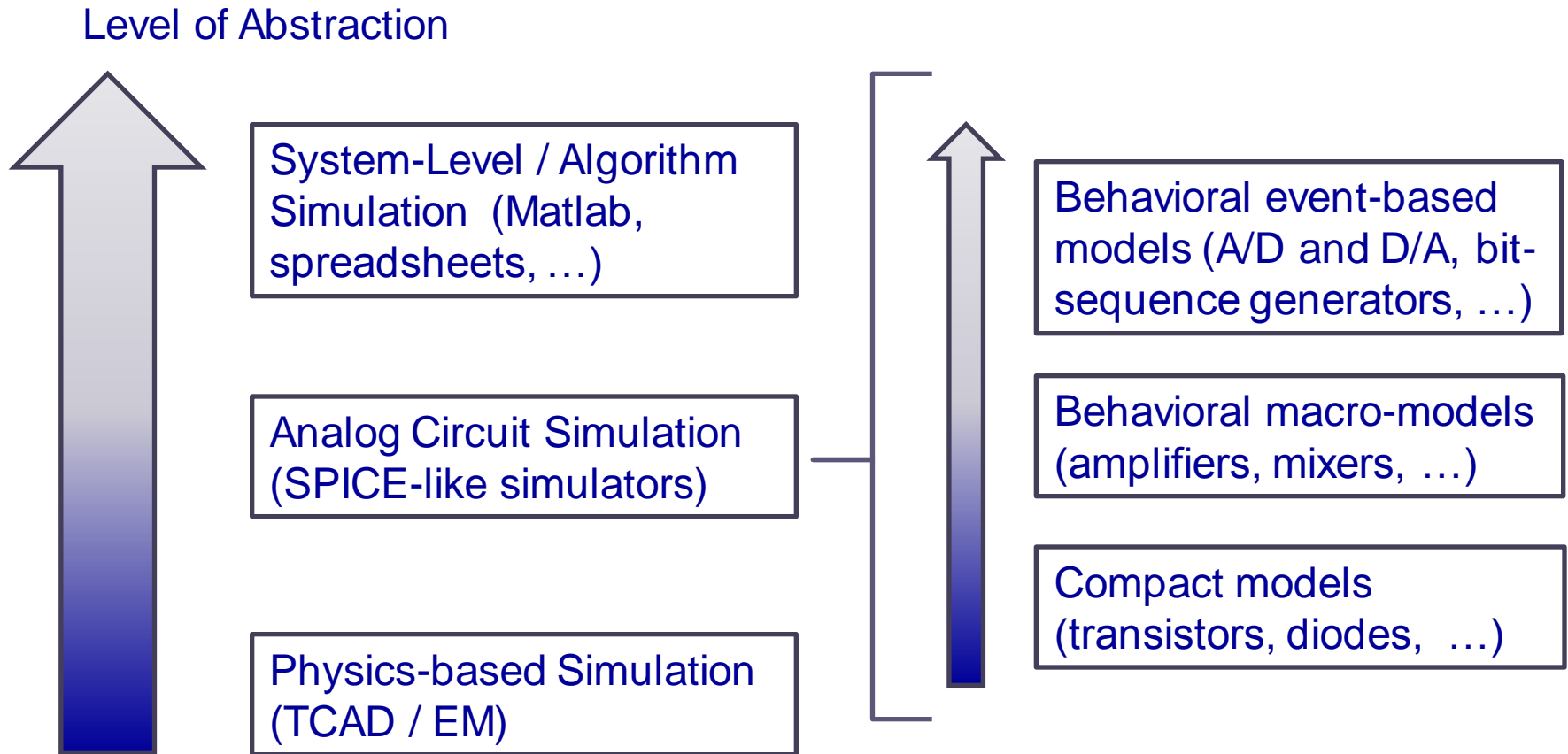
**IEEE Custom Integrated Circuits Conference,
San Jose, CA, Sep. 2011**

Tiburon Design Automation, Inc.
www.tiburon-da.com

Outline

- Introduction / Motivation
- Verilog-A Language Overview and Tutorial
- Selected Topics
 - Noise (including correlation and coloring)
 - State Variables and Modified Nodal Analysis
 - The ddx() operator
 - Restrictions for RF simulation
- Compact Modeling
- Higher-Level (Behavioral) Modeling
- External Resources

Simulation and Modeling



Circuit Simulators and Analog Models

- Traditionally, analog circuit simulators shipped with a fixed set of built-in models
 - Simulator-specific
 - Written in C or Fortran
 - Not accessible to users for modification or inspection
 - Some higher-level models (macromodels) may have been provided as netlist fragments or libraries
- Over time, interfaces for custom user-defined models were introduced
 - Custom C-based interfaces
 - Custom symbolic interfaces

Analog Model Interfaces (Before AHDLs...)

- C-Based Model Interfaces
 - Proprietary
 - Typically simulator specific
 - Burden on model developer to
 - Hand-calculate derivatives
 - (Sometimes) write analysis-specific code
 - Handle software engineering details
- Custom Symbolic Interfaces
 - Non-standard
 - Simulator specific
 - Limited capabilities

Analog Hardware Description Languages

- Analog Hardware Description Languages (AHDLS) provide important benefits:
 - Ease of development
 - Automatic computation of derivatives
 - Language specifically designed for analog modeling
 - Model portability
 - Across different simulators
 - Across various analysis types
 - Suitable for full range of analog model types
 - Behavioral level down to transistor level
- Gaining increasing acceptance
 - Available in most commercial circuit simulators today

Verilog-A...

- A portable, standard, widely supported analog hardware description language
- Applicable across the full range of analog modeling tasks
 - Compact (transistor-level) models
 - High-level analog behavioral models
 - Signal-flow modeling
- Supports behavioral modeling constructs
 - Event-triggered operators (crossing, timer)
 - delay, slew, transition, laplace, integration, etc.
- Provides access to file I/O and to various simulator facilities

A Verilog-A Code Snippet

```
module simple_diode(pos, neg);  
inout pos, neg;  
electrical pos, neg;  
  
parameter real Area = 1.0 from (0:inf);  
parameter real Is=1e-14, n = 2, Cjo=0;  
parameter real Phi = 0.7, m_ = 0.5, tt = 1p;  
real Id, Qd, Vd;  
  
analog begin  
    Vd = V(pos, neg);  
    Id = Area*Is*(exp(Vd/(n*$vt)) - 1);  
    Qd = tt*Id + Area*Vd*Cjo/pow((1-Vd/Phi), m_);  
    I(pos, neg) <+ Id + ddt(Qd);  
  
end  
endmodule
```

port declarations
and disciplines

parameter
and variable
declarations

analog block

Verilog-A Modules

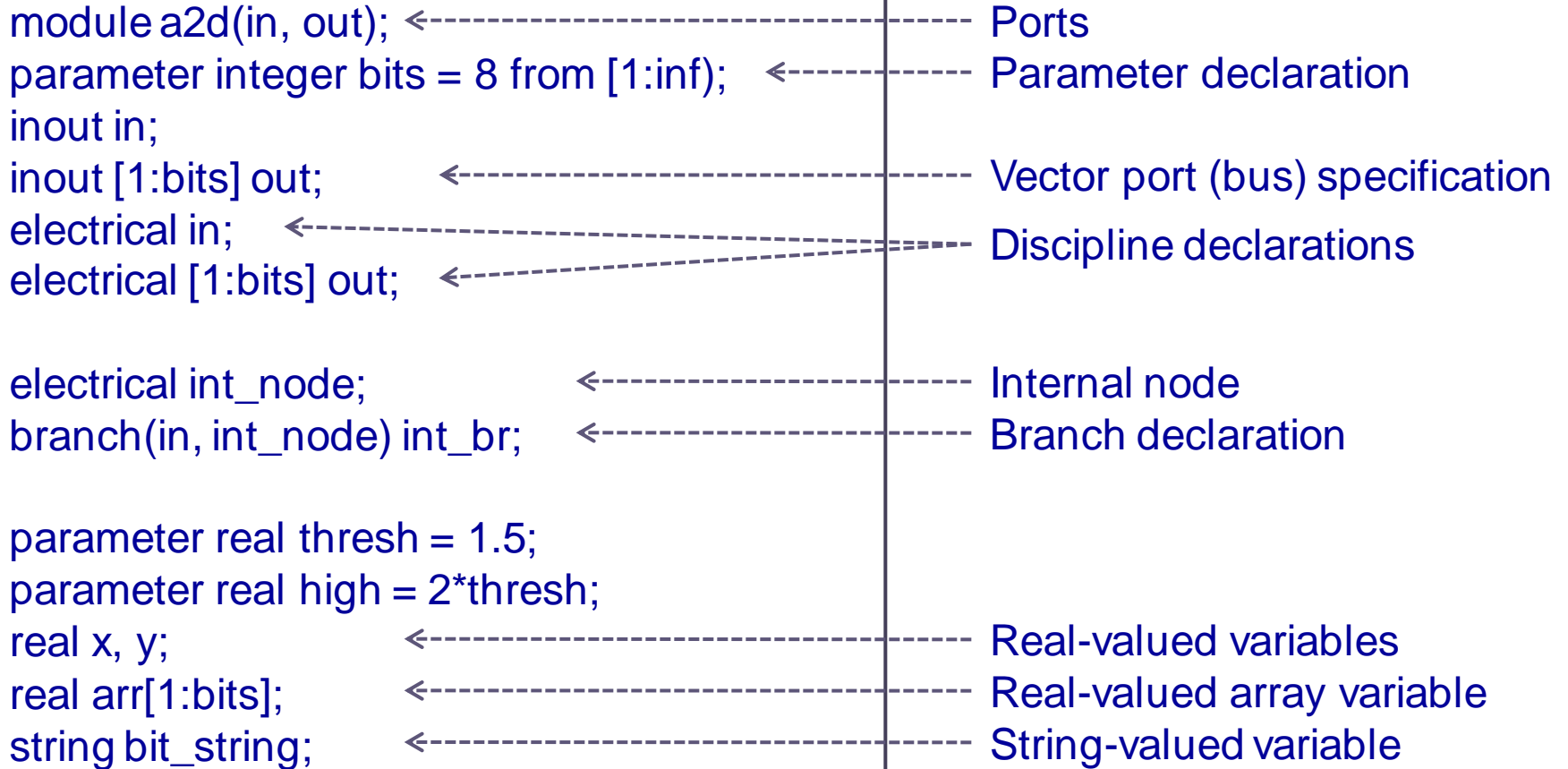
A Verilog-A module contains some combination of:

- Ports that interface to the external world
- Internal nodes local to the module
- Parameter and variable declarations
- Analog block(s) that specify the module's electrical behavior
- Analog functions that can be called from within the module
- Hierarchical structure (to instantiate circuit elements or other modules)

Verilog-A Data Types

- Parameters
 - real, integer, string
 - array of the above
- Variables
 - real, integer, string
 - array of the above
- Nets
- Branches
- Genvars

Verilog-A Data Types (cont.)

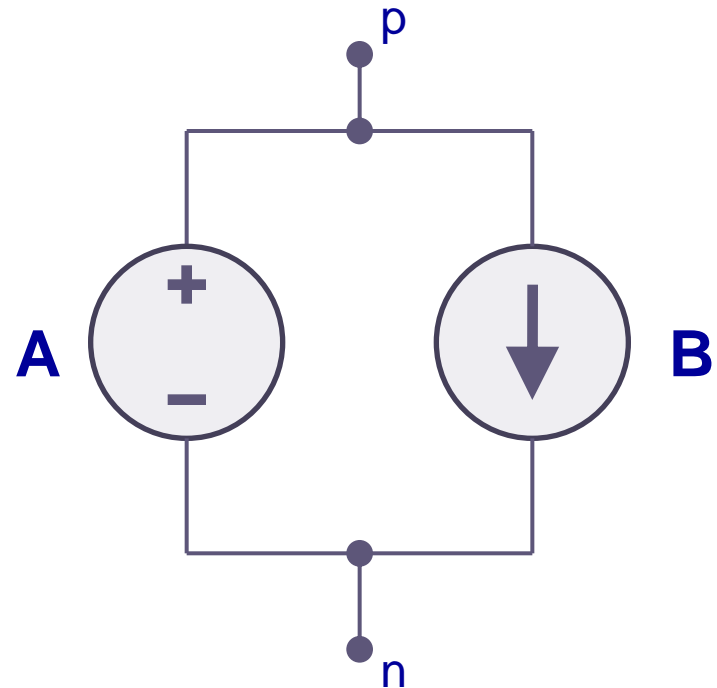


Source Branches

Any branch assigned by a contribution statement is a “source branch”:

electrical p, n;
branch (p, n) br;

$V(p, n) <+ A$;
 $I(\text{br}) <+ B$;

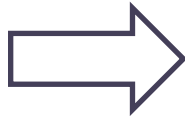


Source branches may be probed (accessed on the right-hand-side) for potential or flow.

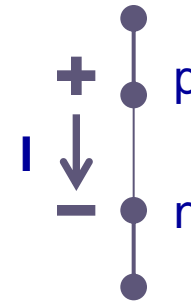
Probes

Any branch **not** assigned by a contribution but accessed (on the right-hand-side) is a “probe.”

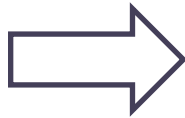
$$x = I(p, n);$$



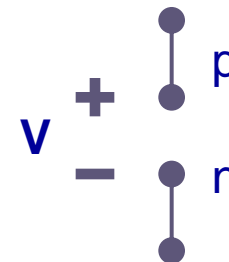
Short nodes p and n;
measure current



$$x = V(p, n);$$

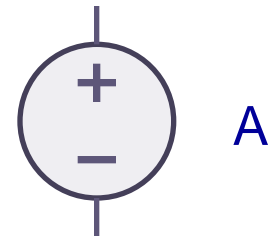
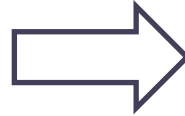


measure voltage

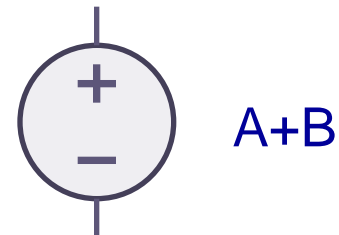
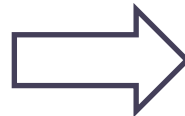


Contributions and Value Retention

$V(p, n) \leftarrow + A;$

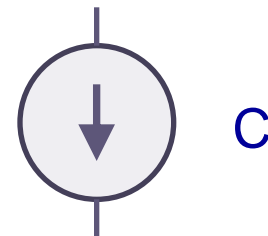
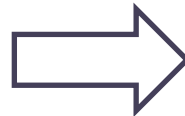


$V(p, n) \leftarrow + B;$



----- Branch switches; contributions discarded -----

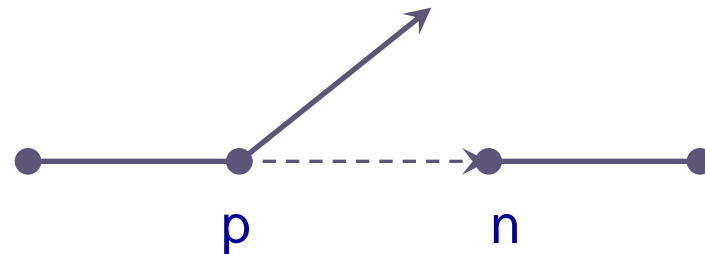
$I(p, n) \leftarrow + C;$



Switch Branches

Branches can “switch” dynamically between flow and potential:

```
if(V(open) < thresh)
  V(p, n) <+ 0; // short
else
  I(p, n) <+ 0; // open
```



- Can introduce discontinuities
- Can be problematic from the standpoint of convergence and performance
- Sometimes inadvertently introduced via programming errors

Verilog-A Statements

- Variable assignments
- Contribution statements
- Conditional statements (if/then/else)
- Looping constructs
 - for
 - while
 - repeat
- Input/Output Statements (C-like)
 - textual display (\$strobe, \$debug, ...)
 - file handling (\$fopen, \$fstrobe, ...)

Analog Block

- Executed on every iteration during analysis
- Computes quantities to be contributed and/or displayed
- Determines the module's intrinsic I-V characteristics
- Is the only type of block that can contribute to an analog net

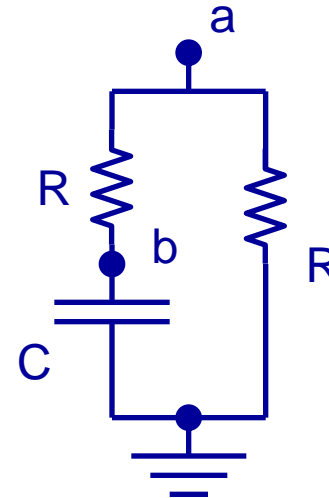
Analog Initial Block

- Executed once at the beginning of each analysis
 - Re-executed for parameter sweeps (e.g., swept-param DC)
- Useful for initialization
 - Compute parameter dependent constants
 - Check static (parameter-dependent) error/warning criteria
 - Initialize file I/O (e.g., \$fopen)
- May not contain:
 - Branch access
 - Contribution statements
 - Analog operators
 - Analog events

Hierarchy

Modules may hierarchically instantiate instances of other modules or circuit elements:

```
parameter R = 50.0 from (0:inf);  
parameter C = 1p from (0:inf);  
electrical a, b, gnd;  
ground gnd;  
  
resistor #(.r(R)) r1(a, b), r2(a, gnd);  
capacitor #(.c(C)) c1(b, gnd);
```



Analog Operators

- `ddt()`
- `idt()`
 - And its offshoots: `idt` with `ic`, `idtmod()`, `idtmod()` with `assert`, etc.
- `absdelay()`
- Laplace family of operators
 - `laplace_nd`, `laplace_np`, `laplace_zd`, `laplace_zp`
- `transition()`
- `slew()`
- Z-transform filters

Analog Events

- “Global” events
 - @(initial_step(...))
 - @(final_step(...))
- “Monitored” events
 - @(timer(...))
 - @(cross(...))
 - @(above(...))
- Event-or'ing:
 - @(timer(T) or cross(V(trigger)))

“analog initial” vs. “@(initial_step)”

```
analog initial begin
  area = 0.5*w*h;
  if(area < minArea || area > maxArea)
    $strobe("Are you sure? Area=", area);
  fd = $fopen("my_file.txt", "w");
end
```

One-off parameter-dependent
computations / diagnostics

File handle initialization

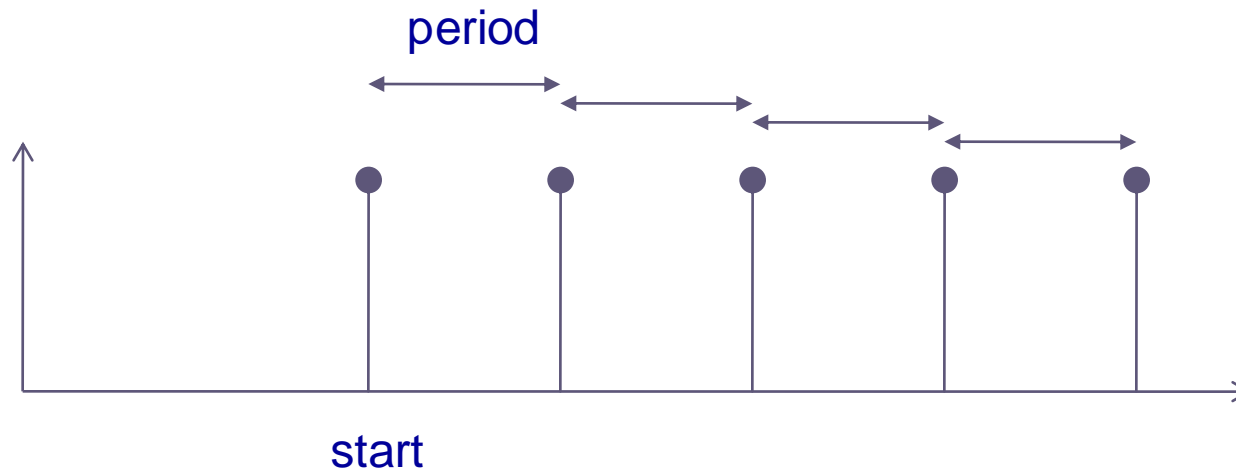
```
analog begin
  @(initial_step("static")) begin
    $fstrobe("Initial val:", V(in));
    $debug(V(in), V(out));
  end
end
```

Branch access

Bias-dependent diagnostics

Timer

```
@(timer(start, [period], [tol], [enable]))  
  <body>
```

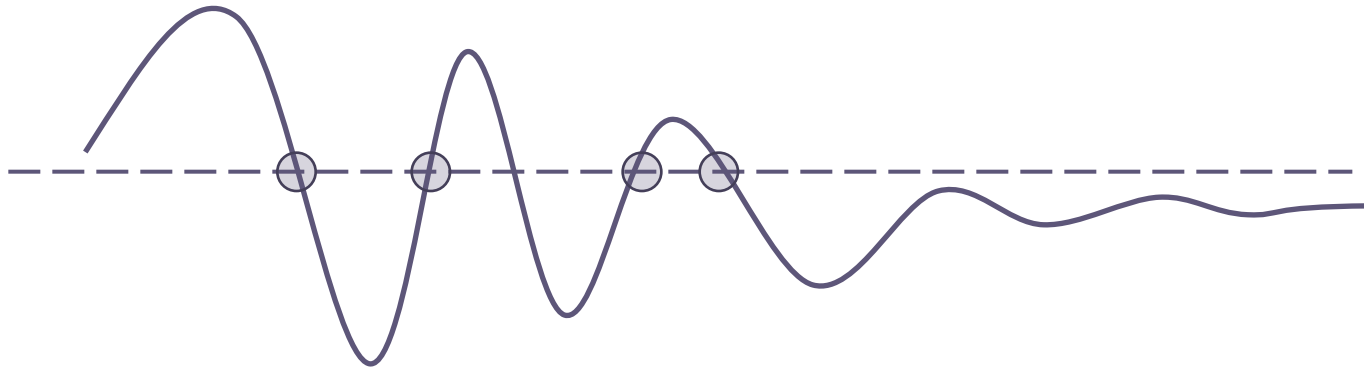


Period is optional

<body> is executed every time the timer fires

Cross

@(cross(expr, [dir], [time_tol], [expr_tol], [enable]))
<body>



<body> is executed every time a crossing occurs

dir == 0: either positive or negative crossings (default)

dir == 1: positive crossings only

dir == -1: negative crossings only

Analog Functions

```
analog function real softexp;  
real x;  
input x;  
begin  
    if (x < `maxarg)  
        softexp = exp(x);  
    else  
        softexp = (x + 1 - `maxarg)*exp(`max_arg);  
end  
endfunction
```

Analog function
definition

```
analog begin  
    I(a, c) <+ Is*softexp(V(a,c)/$vt - 1);  
end
```

Analog function
usage

The Verilog-A Simulation Flow

1. Start with guess x_0 (or previous solution, if available)
2. Reset all variables, file-descriptors, and state to last-accepted value
3. Evaluate Verilog-A device(s) based on solution
 - Compute residuals, Jacobian matrix elements
 - Execute \$debug statements
 - Update analog operator and event state
4. If not converged, go to step 2
5. If converged, trigger “accept”
 - Accept variables values and analog state
 - Execute \$strobe statements
6. Advance to next point (next time point in transient, etc.)

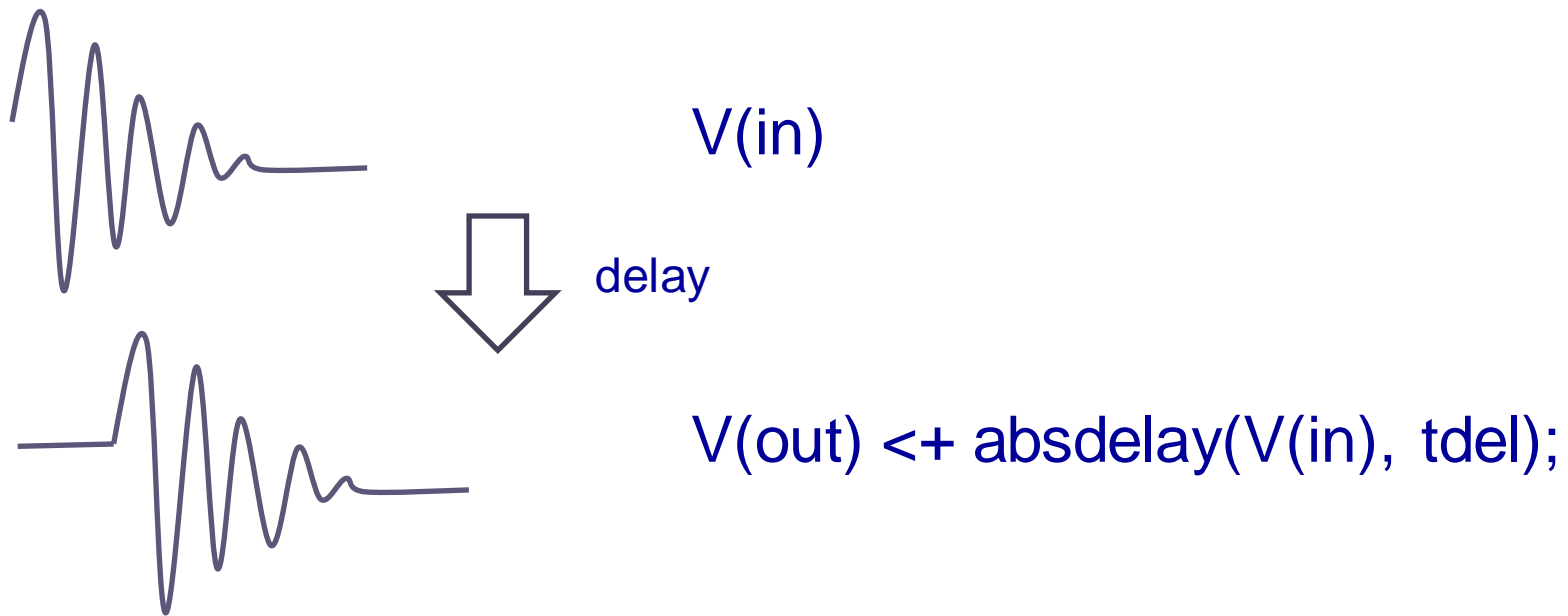
Variables and Memory States

- All variables are initially set to zero
 - String variables are set to “empty string”
- On “accept,” values are permanently retained (accepted)
- If a variable is used in the analog block before it is set, its value is taken to be the previously accepted value

```
analog begin
    real t, tprev, dt;
    t = $abstime;           // t is not a memory state
    dt = t - tprev;        // tprev is a memory state
    $strobe("time step == ", dt);
    tprev = $abstime;      // will be the "accepted" value
end
```

Operators, Events, and Internal State

A key defining characteristic of analog operators and events is their need to store “history,” or “state”:



Static Contexts

Objects with state can only exist inside static contexts:

```
parameter enable=1;  
if(enable)  
    @(cross(V(clk))) out = !out;
```

```
if(V(en) > 0)  
    @(timer(0, T)) count = count+1;
```

```
if($abstime > 1n)  
    x = idt(y, ic);
```

```
@(timer(1n, 1p))  
    x = absdelay(y, 1n);
```

Legal: “enable” is a parameter

Illegal: analog operators and events cannot appear inside non-static (time-varying or bias-dependent) constructs

Looping Contexts

Objects with state are not allowed within ordinary loops:

```
integer i;  
for(i = 0; i < N; i= i+1) begin  
    V(out[i]) <+ transition(x[i]);  
end
```

Illegal! Transition carries internal state – do we mean 1 transition, or N copies of it?

```
integer i;  
for(i = 0; i < N; i= i+1) begin  
    @(cross(V(trig[i])))  
    latch[i] = V(in[i]);  
end
```

Illegal! Cross carries internal state – we need to explicitly specify that we need N copies.

Genvars and Genvar Loops

The “genvar” data type: a Verilog-specific variable type for “unrolling” structural for-loops:

- Integer-valued
- Used in loop bounds expressions
- Loop bounds must be static
 - Expressions can consist of parameters, constants, and other genvar variables
- May not be assigned anywhere outside of loop-control expressions

Genvar Loops and Analog Events/Operators

```
genvar g;  
parameter integer K=5;  
for(g = 0; g <= K; g = g+1) begin  
    V(out[g]) <+ transition(x[g]);  
end
```

V(out[0]) <+ transition(x[0]);
V(out[1]) <+ transition(x[1]);
⋮ K+1 copies
V(out[K]) <+ transition(x[K]);

```
for(g = 1; g <= K; g = g+2) begin  
    @(timer(0, g*g)) out[g] = !out[g];  
end
```

@(timer(0, 1*1)) out[1] = !out[1];
@(timer(0, 3*3)) out[3] = !out[3];
⋮
@(timer(0, K*K)) out[K] = !out[K];

Dynamically Enabling/Disabling Events

```
if($abstime < 10*T)
  @(timer(0, T)) count = count+1;
```

```
if(V(clk_enable) > thresh)
  @(cross(V(clk)) vsample = V(in));
```

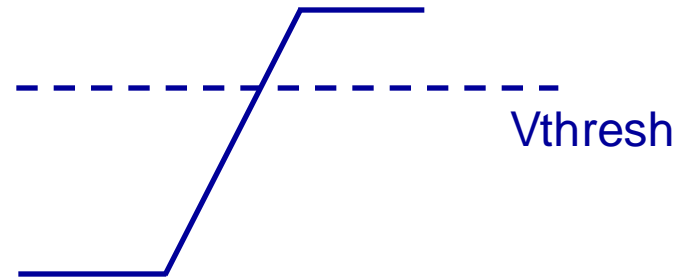
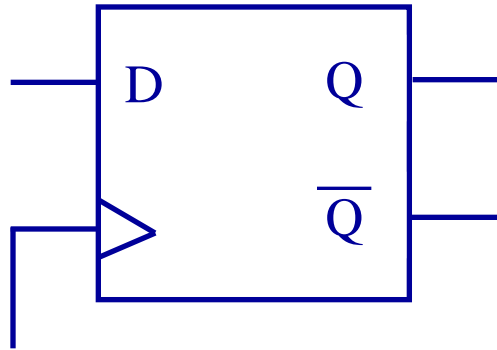
Illegal: events cannot be dynamically enabled/disabled via conditionals

LRM 2.3 has added enable options directly to the timer, cross, and above events:

```
@(timer(0, T, , $abstime < 10*T)) ...
```

```
@(cross(V(clk), , , V(clk_enable) > thresh) ...
```

Memory States and Events



```
@(cross(V(clk)-vthresh, 1)
    x = V(din);
```

```
V(Q) <+ transition(A*(x > vthresh), delay, riseTime, fallTime);
V(Qbar) <+ ...
```

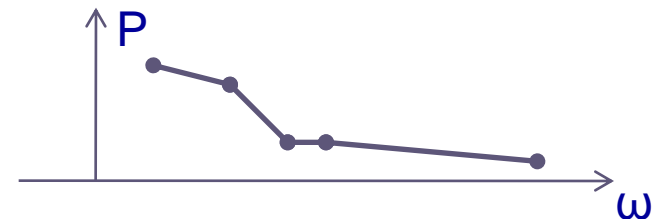
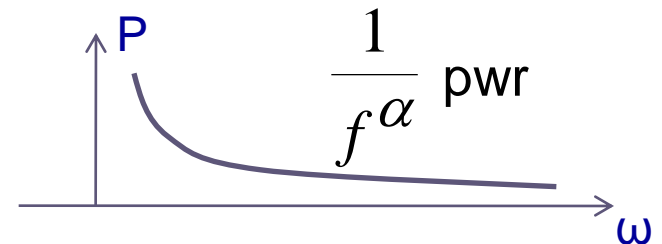
Noise Sources

Small-signal noise sources in Verilog-A:

```
x = white_noise(pwr, name);
```

```
y = flicker_noise(pwr,  $\alpha$ , name);
```

```
z = noise_table(array, name);
```



Small-Signal Noise Examples

Resistor: Thermal noise



```
I(p, n) <+ V(p, n)/R + white_noise(4*`P_K*$temperature/R);
```

Diode: Thermal noise and flicker noise

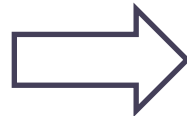


```
I(a, c) <+ Idiode + white_noise(2*`P_Q*Idiode) +  
flicker_noise(kf*pow(abs(Idiode)), af), ef);
```

Noise Sources and Correlation

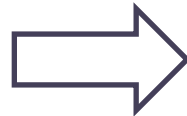
In Verilog-A, each noise source entity is independent:

```
x1 = white_noise(P1);  
x2 = white_noise(P2);
```



x1 and x2 independent (by definition)

```
V(n1) <+ x1;  
V(n2) <+ x2;
```



V(n1) and V(n2) are uncorrelated

```
V(n3) <+ K1*x1 + K2*x2;
```



Correlated with V(n1) if $K1 \neq 0$
Correlated with V(n2) if $K2 \neq 0$

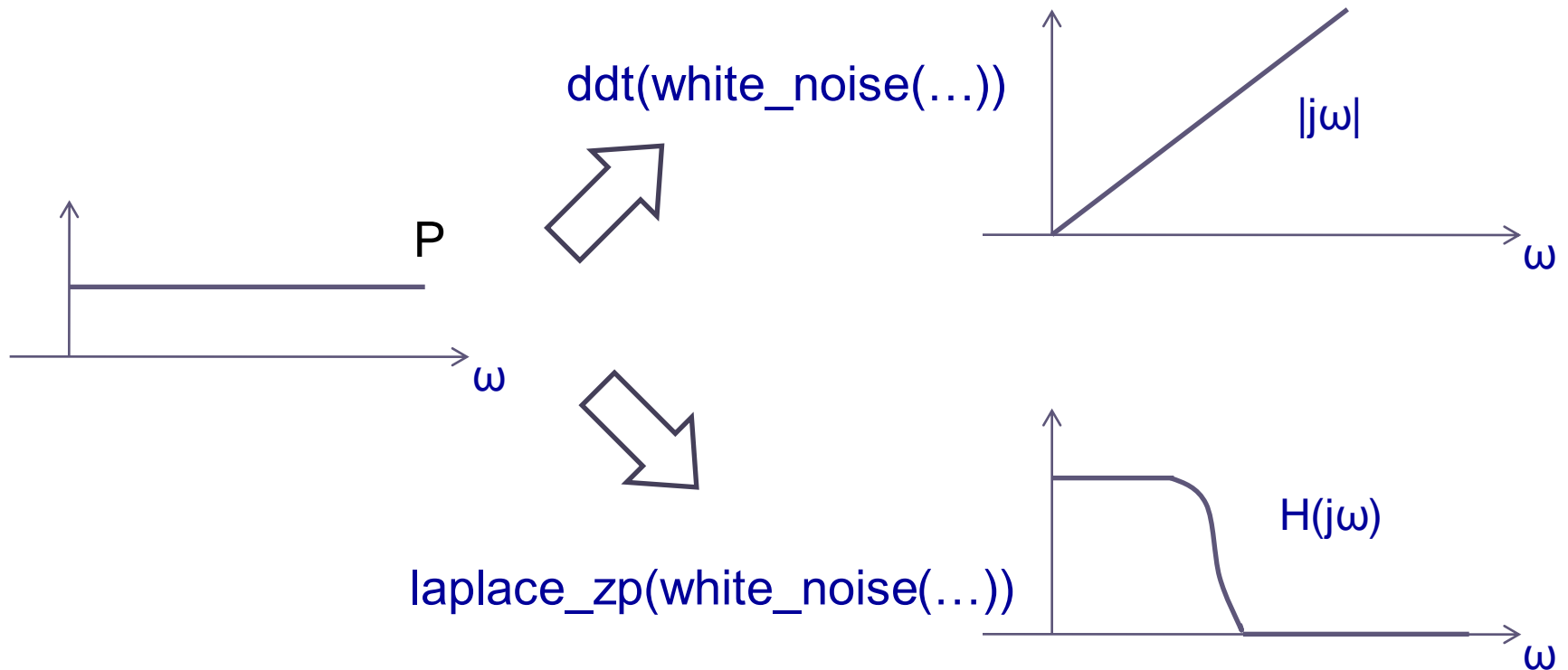
In Matrix Form...

```
x1 = white_noise(P1);  
x2 = white_noise(P2);  
V(n1) <+ x1;  
V(n2) <+ x2;  
V(n3) <+ K1*x1 + K2*x2;
```

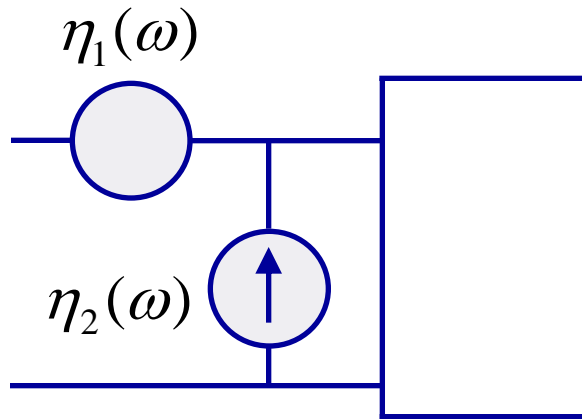
$$\begin{bmatrix} \langle n_1 n_1^* \rangle & \langle n_1 n_2^* \rangle & \langle n_1 n_3^* \rangle \\ \langle n_2 n_1^* \rangle & \langle n_2 n_2^* \rangle & \langle n_2 n_3^* \rangle \\ \langle n_3 n_1^* \rangle & \langle n_3 n_2^* \rangle & \langle n_3 n_3^* \rangle \end{bmatrix} = \begin{bmatrix} P_1 & 0 & K_1 P_1 \\ 0 & P_2 & K_2 P_2 \\ K_1^2 P_1 & K_2^2 P_2 & K_1^2 P_1 + K_2^2 P_2 \end{bmatrix}$$

Colored Noise

Analog operators may be applied to color noise sources:



Combining Correlation and Coloring

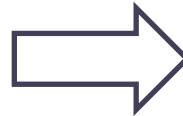


Correlation

A = white_noise(K)
B = white_noise(P1-K)
C = white_noise(P2-K)
n1 = A+B
n2 = A+C

Frequency Dependence

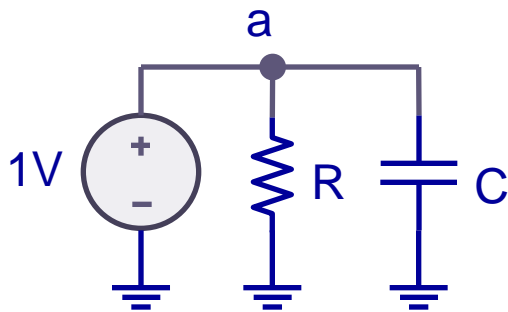
x = white_noise(...)
xc = laplace_zp(x, zeros, poles)



General N-Port Noise
Representation

Analog State Variables and Modified Nodal Analysis (MNA)

Verilog-AMS standard explicitly refers to Modified Nodal Analysis:



Kirchoff's Current Law (KCL) at node a:

$$V(a)/R + C \cdot \text{ddt}(V(a)) + I_{\text{src}} = 0$$

Branch equation for voltage source:

$$V(a) = 1V$$

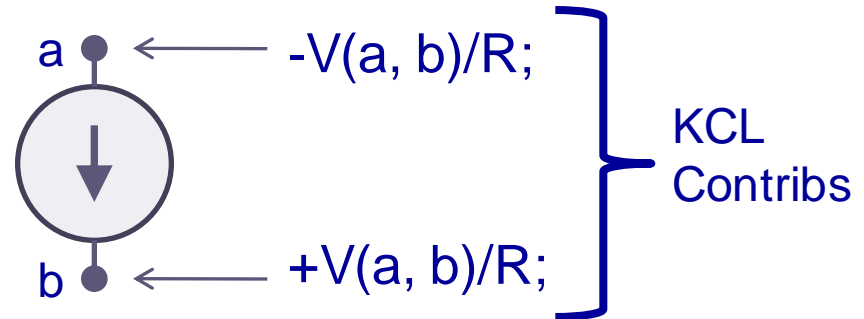
Current controlled elements (e.g., voltage sources, inductors) introduce an extra state variable (current through device).

Analog state variables are the set of node voltages and branch currents.
In this case: $V(a)$ and I_{src} .

Analog State Variables and Verilog-A

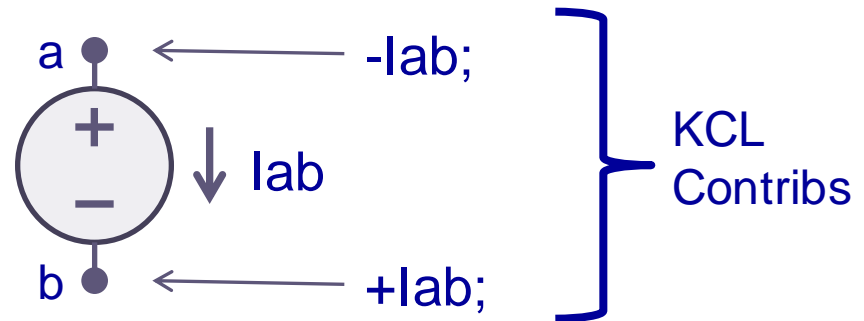
$$I(a,b) <+ V(a, b)/R;$$

No extra branch equation needed



$$V(a,b) <+ R*I(a, b);$$

- Introduce **extra variable** i_{ab}
- Introduce **extra equation**:
 $V_a - V_b - i_{ab} * R = 0$



State Variables and Branches in Verilog-A

In general, introduction of an extra MNA state variable can occur when

- a voltage (potential) is contributed to (left-hand side):
 $V(p, n) <+ \dots;$
- a current (flow) is sensed (right-hand-side):
 $\dots <+ I(p, n);$
- an analog operator like `idt()` is used
- the `ddt ()` operator is used in a “non-trivial algebraic context” (more on this later)

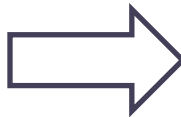
State Variables and Ordinary Variables

State variables are varied by the simulator until convergence is reached:

```
electrical a, b;  
real x, y;
```

```
V(a) <+ V(b);  
$debug( V(a) - V(b) );
```

```
x = V(b);  
$debug( V(b) - x );
```



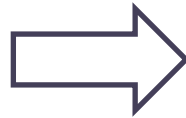
- $| V(a) - V(b) | < \epsilon_{tol}$
at **convergence**
- \$debug may print nonzero values

- x is precisely equal to V(b)
- \$debug will print 0.0

Consequences for Floating Point Expressions

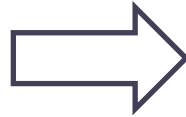
Consider:

```
x = exp(V(a));  
y = ln(1 + x);
```



x and y guaranteed to always be positive

```
V(b) <+ exp(V(a));  
y = ln(1 + V(b));
```



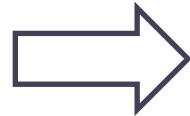
y may take on negative values during convergence process

```
x = 1.0;  
y = y / x;
```



Safe: $x \neq 0.0$

```
V(b) <+ 1.0;  
y = y / V(b);
```



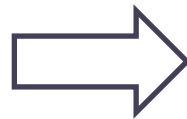
Unsafe: V(b) may be zero

Whenever possible, this should be taken into account when formulating floating-point expressions.

The ddx() Operator

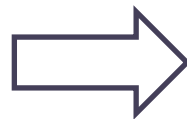
The ddx() operator computes the symbolic derivative of an expression with respect to a state variable:

```
x = V(in);  
V(out) <+ a0*(x + a1*(x + a2*x));  
gain = ddx(V(out), V(in));
```



gain =
 $a_0 + 2*a_0*a_1 + 3*a_0*a_1*a_2$

```
module bjt_intrinsic(c, b, e);  
....  
I(c, e) <+ Ic;  
gm = ddx(Ic, V(b, e));
```



transconductance
calculation

The ddx() Operator (cont.)

- General form is $\text{ddx}(\text{expr}, \text{state_var})$
 - expr is differentiated with respect to state_var
- state_var must be a single state variable
 - Not a two-node unnamed branch
 - Not an ordinary variable
- The derivative is a partial derivative, with each state variable viewed as an independent (free) variable

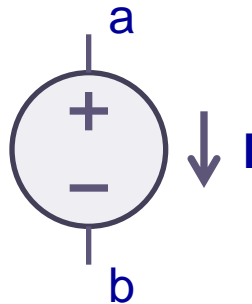
State Variables and the ddx() Operator

If voltage is contributed or current is probed, the state variables are:

$V(a)$

$V(b)$

$I(a,b)$



Note: $V(a, b) = V(a) - V(b)$
when used as a target in ddx

The rules for voltage (potential) do not apply to current (flow):

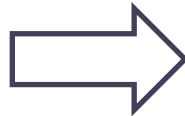
$I(a, b)$: current through unnamed branch between nodes a and b

$I(a)$: current through unnamed branch between node a and ground

$I(b)$: current through unnamed branch between node b and ground

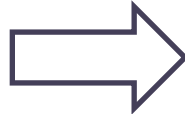
State Variables and the ddx() Operator

```
result = ddx(V(a,b), V(a));
```



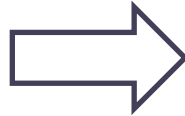
result = 1.0; $V(a,b) = V(a)-V(b)$

```
result = ddx(V(a,b), V(b));
```



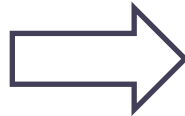
result = -1.0; $V(a,b) = V(a)-V(b)$

```
result = ddx(V(a,b), V(a,b));
```



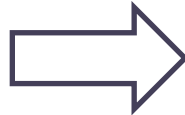
illegal: $V(a,b)$ is not a state variable

```
result = ddx(I(a,b), I(a,b));
```



result = 1.0

```
result = ddx(I(a,b), I(a));
```



result = 0.0; $I(a,b)$ and $I(a)$ are independent state variables

```
result = ddx(I(a,b), I(b));
```



result = 0.0; $I(a,b)$ and $I(b)$ are independent state variables

Common ddx() Misconceptions

```
V(b) <+ V(a);  
result = ddx(V(b), V(a));
```



result = 0.0; V(a) and V(b) are independent state variables

```
real var;  
var = V(a);  
result = ddx(var, V(a));
```



result = 1.0; var is a real variable, not a state variable

```
real var;  
var = V(a) + V(a, b);  
result = ddx(var, V(a));
```



result = 2.0, **not** 1.0.

Compact Model Development in Verilog-A

- Verilog-A is emerging as the leading AHDL for compact model development
 - Available across a wide range of simulation platforms
 - Has specific extensions for compact modeling
 - Endorsed as a release language by Compact Model Council
 - PSP and BSIMSOI models are being released in Verilog-A
 - A large number of compact transistor models have been implemented in Verilog-A:
 - PSP, BSIMSOI, BSIM3, BSIM4, BSIM5, MOS 9, MOS11, HiSIM
 - Gummel-Poon BJT, VBIC, HICUM, Mextram 504
 - Parker-Skellern, Anvelov, Curtice MESFET
 - Many others

Considerations for Compact Model Development

- Minimizing unnecessary state variables and branches
 - Collapsing internal nodes
 - Avoiding voltage contributions and switch branches
- Continuous derivatives
- Charge-conservation
- Compatibility with RF simulation algorithms
 - Memory states
 - Events
 - Analysis-specific constructs
 - Analog operators with hidden state

Collapsible Nodes

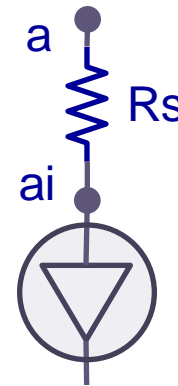
Consider an intrinsic diode with a series resistance R_s :

$$I(a, ai) <+ V(a,ai)/R_s;$$

- ❑ Invalid for $R_s == 0$ (and poor for R_s small)
- ❑ Adds state variable for node ai even if $R_s == 0$

$$V(a, ai) <+ I(a,ai)*R_s;$$

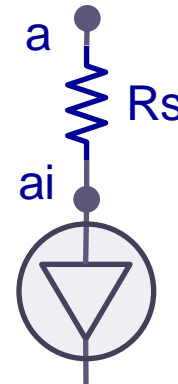
- ❑ Works for all values R_s
- ❑ Always introduces two additional state variables:
 - one state variable for the internal node ai (even if $R_s == 0$)
 - one state variable for the branch equation corresponding to $V(a,ai)$



Collapsible Nodes (cont.)

Verilog-A “idiom” for collapsible nodes:

```
if(Rs > 0)
  I(a, ai) <+ V(a, ai)/Rs;
else
  V(a, ai) <+ 0.0;
```



More generally:

```
if(const_expr)
  I(p, n) <+ expr;
else
  V(p, n) <+ 0.0;
```

const_expr must be a function of parameters and constants (possibly expressed through intermediate variables)

The Collapsible Nodes Idiom

- Applicable to nonlinear resistors (e.g., base resistance in BJTs)
 - `if(Rbm == 0)`
 `V(b, bi) <+ 0.0;`
 `else`
 `I(b, bi) <+ nonlin_expression_for_lb`
- Some common errors to avoid:
 - The conditional expression must be static (ie, a function of parameters, constants, and/or variables that in turn depend only on parameters and constants)
 - The `I(...)` and `V(...)` contributions on the LHS must refer to the same branch

Avoiding Branch-ddt Equations

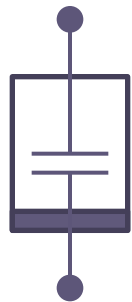
MNA: $f(x(t)) + \frac{d}{dt} q(x(t)) = u(t)$

$I(a, b) \leftarrow \text{ddt}(Q(a, b));$

$V(a, b) \leftarrow \text{ddt}(\text{Phi}(a, b));$

Ordinary capacitors and inductors (both linear and nonlinear) naturally fit into this framework without extra branch equations.

Branch-ddt Equations (cont.)



$Q(v)$

$$I(v) = \text{ddt}(Q(v)) = Q'(v) * \text{ddt}(v)$$

$$I(p, n) <+ \text{ddt}(Q(p, n))$$

robust and efficient

$$I(p, n) <+ C(V(p, n)) * \text{ddt}(V(p, n));$$

problematic and inefficient

Capacitive formulation

- Results in additional equation / state-variable
- Fails to conserve charge (in a numerical sense)

Internally, simulator will add a new state variable x_{br} and a new equation:

$$I(p, n) <+ C(V(p, n)) * x_{br};$$
$$\text{ddt}(V(p, n) - x_{br}) == 0$$

Portability Across Analysis Types

- Certain language constructs are not supported by RF analyses (e.g, Harmonic Balance, Shooting, Envelope)
- Should be avoided for reasons of
 - portability
 - consistency across analyses
 - efficiency
- Typically not required (or desired) for compact models

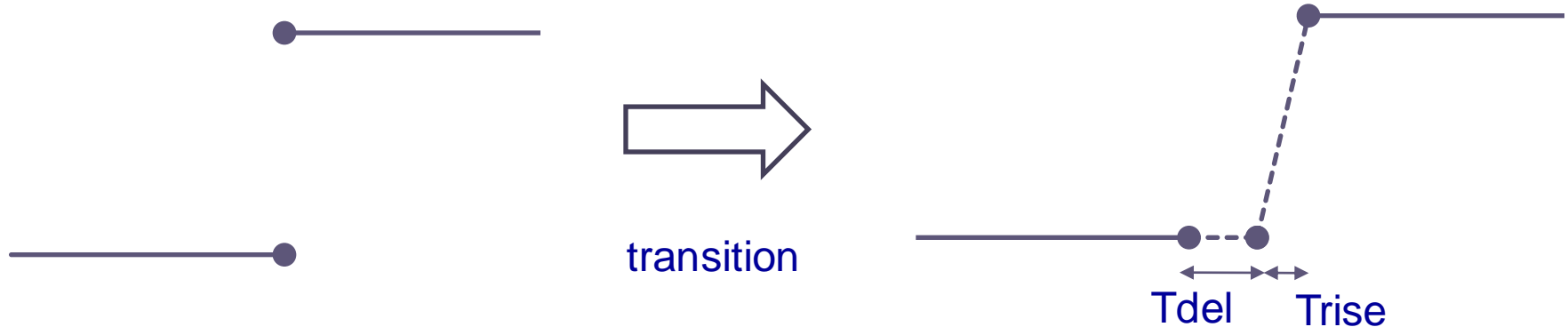
RF Restrictions

- Monitored events (timer, crossing, above)
- Explicit use of time (\$abstime)
- Analog Operators
 - Allowed:
 - Differentiation (ddt)
 - Integration (idt) without initial conditions
 - Allowed, but can result in a performance penalty:
 - Delay
 - Laplace
 - Others are
 - Not safe for RF analysis
 - Not (typically) useful for compact modeling

Behavioral Modeling

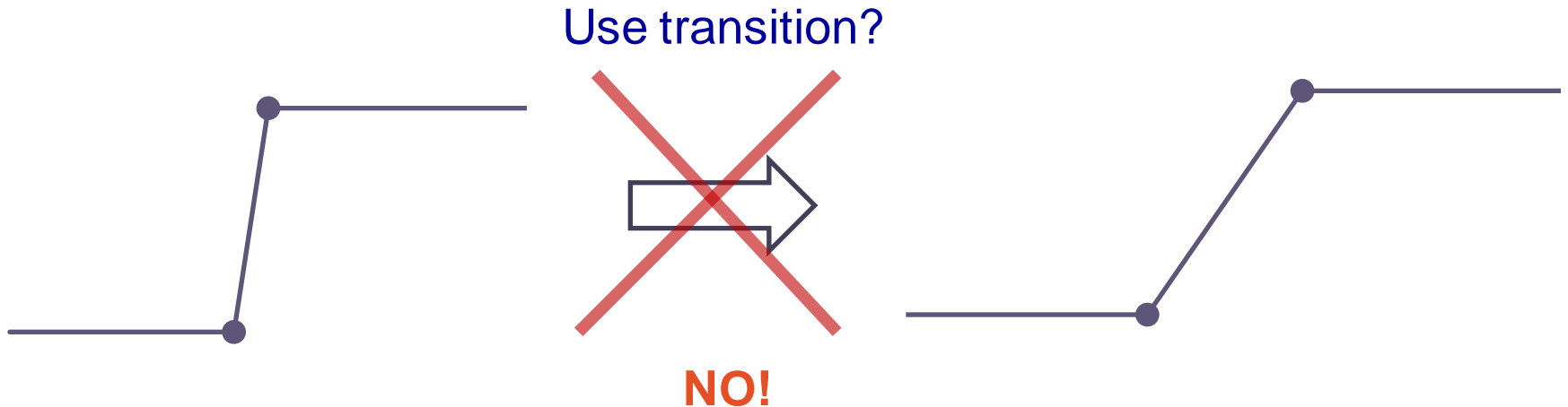
- Preserving Continuity
- Improper Use of Analog Operators
 - Transition applied to continuous signals
 - `idt()` of arg that can never be driven to zero
 - Invalid arguments to analog operators
- Avoiding Common Source-Probe Formulation Errors
 - Inadvertent switch branches
 - Topology errors
- Genvars and Vector Constructs
 - Discussed previously

Continuity: The transition() Operator



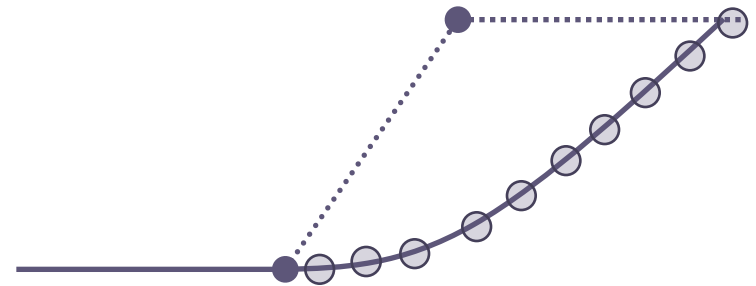
- Turns discontinuous waveforms into piecewise-continuous waveforms
- Input must be strictly quantized (discrete-valued)
- Continuous variation can result in significant performance degradation

Not For Adjusting Slope of Continuous Waveforms...



transition(V(in), 0, tr, tf):

- Interprets any change as the start of a new transition
- Will schedule (and interrupt) multiple transition events for continuous input waveform (even piecewise linear)



Transition Operator Guidelines

- Should be applied to avoid discontinuous contributions
- Should only be applied to quantized signals
- Should not be applied to continuous arguments
 - Any change in argument between two time steps results in a transition being initiated
 - Use slew() or laplace filtering instead
- Should not be applied to expressions which depend continuously on state variables
 - Numerical noise: $\text{transition}(V(a), \dots)$ is still potentially problematic even if $V(a)$ is a piecewise-constant signal

Transition Operator Guidelines (cont.)

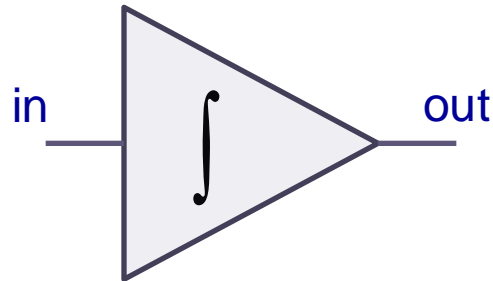
Consider:

```
module smooth_clk(clk, out);  
input clk;  
output out;  
electrical clk, out;  
  
analog V(out) <+ transition(V(clk), 0, 1p);
```

Not advisable!

// Use custom analog function to quantize:
afn_quantize(V(clk))

idt() and Infinite Gain at DC



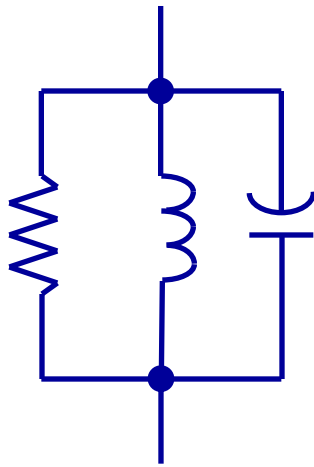
$$V(\text{out}) \leftarrow K \cdot \text{idt}(V(\text{in}));$$

At DC:
$$\int_{-\infty}^0 \text{arg}(t) dt = \infty \bullet \text{arg}(0)$$

For a finite DC result...

- arg(t) must be a function of state variables (or 0.0)
- arg(t) will be forced to zero for t=0

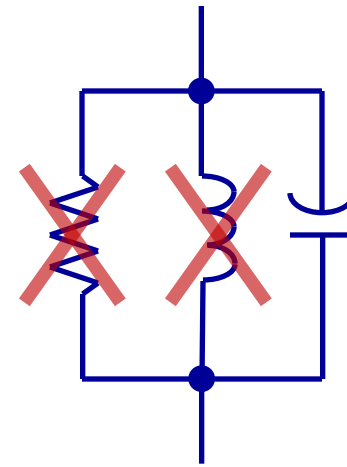
Example of Inadvertent Switch-Branch



Distinct named branches:

electrical a, b;
branch (a, b) brR, brL, brC;

$I(\text{brR}) <+ V(\text{brR})/R;$
 $V(\text{brL}) <+ L * \text{ddt}(I(\text{brL}));$
 $I(\text{brC}) <+ C * \text{ddt}(V(\text{brC}));$



Single unnamed branch:

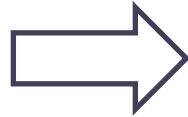
electrical a, b;
 $I(a, b) <+ V(a, b)/R;$
 $V(a, b) <+ L * \text{ddt}(I(a, b));$
 $I(a, b) <+ C * \text{ddt}(V(a, b));$

Inadvertent Short-Circuits (Topology Errors)



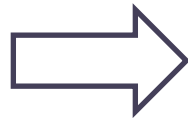
```
module amp(p, n, out);  
...  
V(out) <+ A*V(p, n);
```

```
$strobe( I(<p> ) );
```



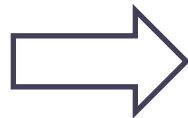
OK: Prints current flowing into pin p

```
$strobe( I(p) );
```



Inadvertent: Shorts node p to ground
[Creates unnamed branch from p to ground]

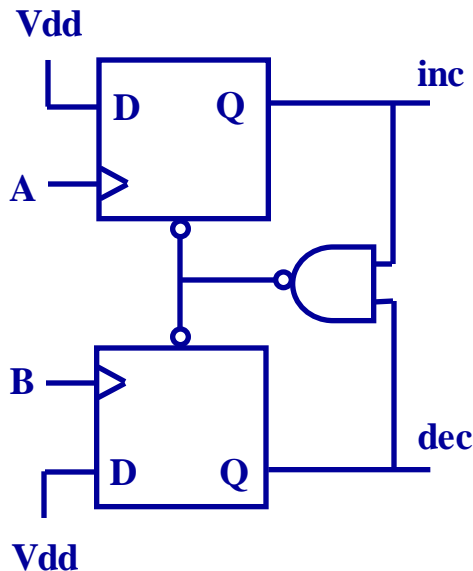
```
$strobe( I(p, n) );
```



Inadvertent: Shorts nodes p and n together
[Creates unnamed branch from p to n]

Example: Simple Phase-Frequency Detector

Circuit may be modeled at various levels:



- ❑ All components described at transistor level (transistors may themselves be Verilog-A based)
- ❑ Flip-flops and nand gate can be handled individually at behavioral level
- ❑ The whole detector circuit can be handled at behavioral level

A Drifting “Clock”

```
module beh_clk(out);  
...  
parameter real freq = 10M;  
parameter drift_frac = 0.0; // Fractional drift  
integer i, outhi;  
real tnext;  
...  
analog begin  
    @(timer(tnext)) begin  
        outhi = !outhi;  
        tnext = tnext + 1.0/(freq*(1 + i*drift_frac));  
        i = i+1;  
    end  
    V(out) <+ transition( outhi ? vhi : vlo, 0, tr, tf);  
end
```

No drift:

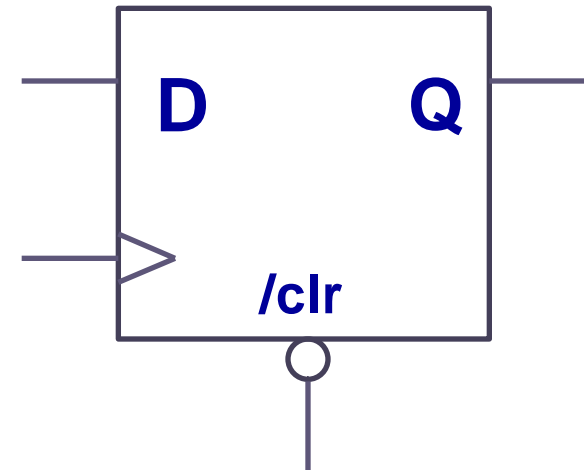


Finite drift:

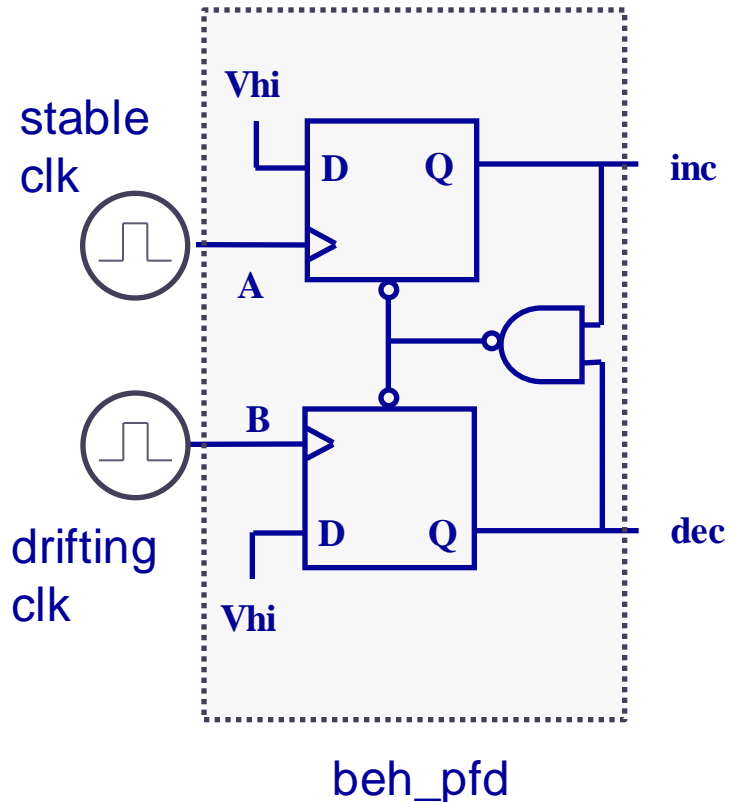


A Simple Behavioral D Flip-Flop

```
module beh_dff(D, clk, Q, Vclr );  
...  
analog begin  
  
    @(cross(V(clk)-vthresh))  
        state = V(D) >= vthresh;  
  
    if(V(Vclr ) < vthresh)  
        state = 0;  
  
    vout = state == 0 ? vlo : vhi;  
  
    V(Q) <+ transition(vout, dly, tr, tf);  
  
end
```



The PFD and a Test Circuit



```
module beh_pfd(A, B, inc, dec, vdd);
```

```
  beh_dff dff1(vdd, A, inc, rst);  
  beh_dff dff2(vdd, B, dec, rst);
```

```
  beh_nand bn(inc, dec, rst);
```

```
endmodule
```

```
Vpwr hi 0 5.0
```

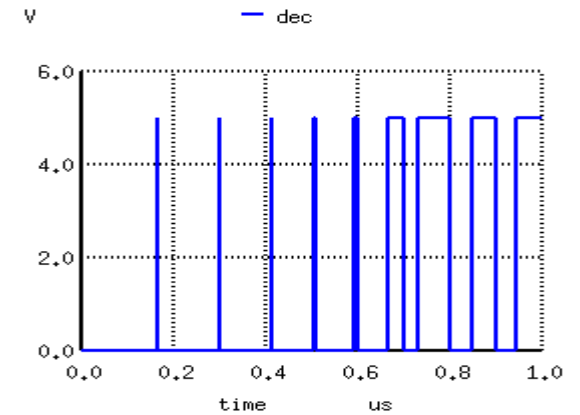
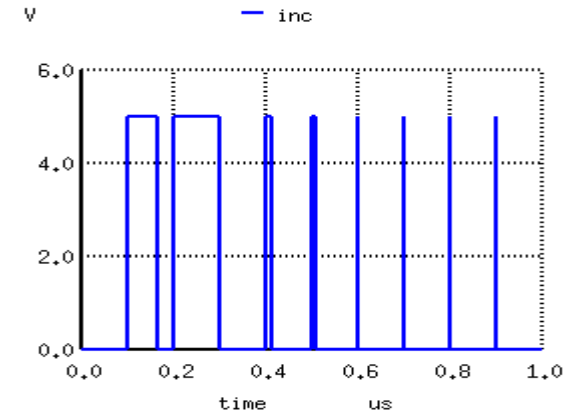
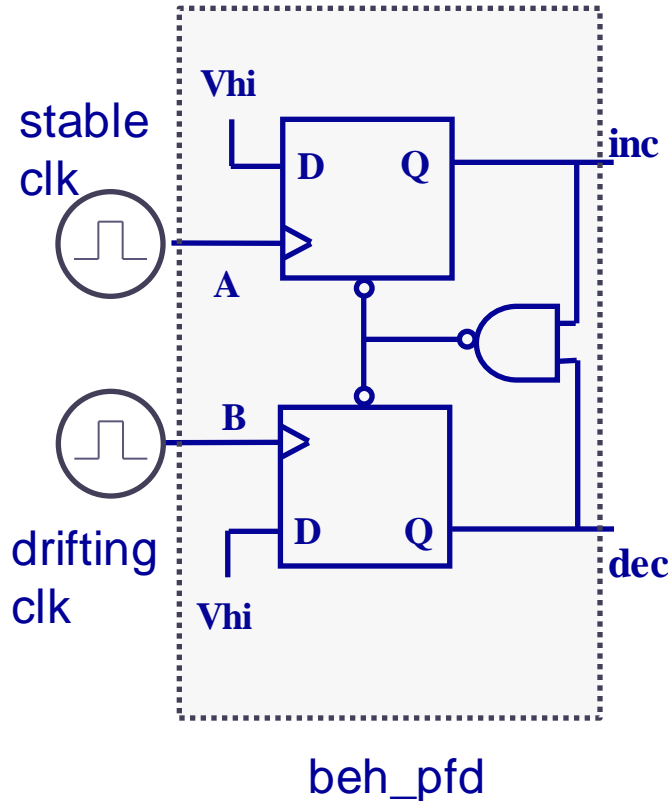
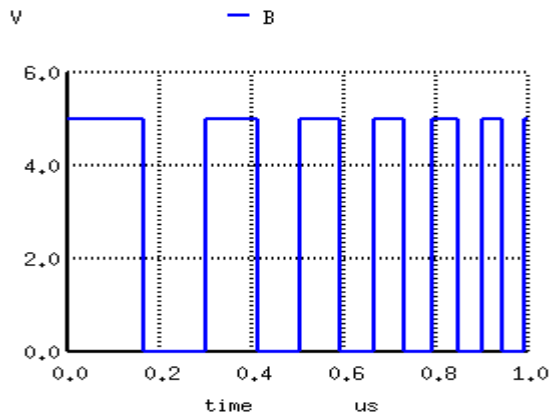
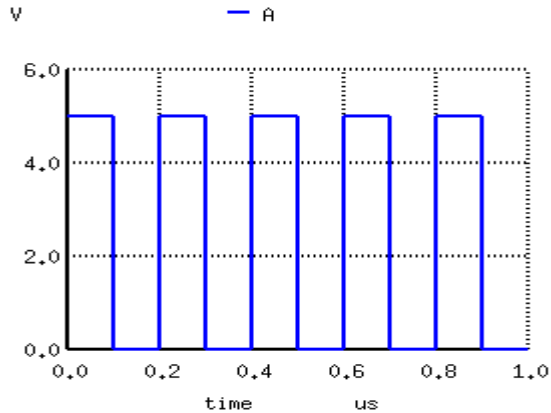
```
Xclk1 A beh_clk freq=10meg
```

```
Xclk2 B beh_clk freq=6meg drift_frac=0.25
```

```
Xpfd A B inc dec hi beh_pfd
```

```
.tran 0.001u 1u
```

Simulation Results



Some External Resources and References

- <http://www.designers-guide.org>
 - Verilog-A and Verilog-AMS models
 - Articles on Verilog-A simulation (hidden state, jitter, behavioral modeling, etc.)
- Compact modeling tutorials
 - “How to (and How NOT to) Write a Compact Model in Verilog-A,”
G. Coram, BMAS 2004
 - “Verilog-A: An Introduction for Compact Modelers,”
G. Coram, MOS-AK/ESSDERC/ESSCIRC Workshop (2006)
- Correlated Noise in Verilog-A
 - “Correlated Noise Modeling and Simulation,”
C. McAndrew et al., NanoTech/WCM 2005