Introduction to Verilog-AMS

Mohamed Watfa

June 4, 2024

Table of Contents

Syntax

The syntax in this document follows these conventions:

- <· | ·>: Choose one item.
- <·>?: Optional item.
- <·>*: Repeat zero or more times.
- <·>+: Repeat one or more times.

Example

```
<integer | real | string> identifier<[range]>? <, identifier<[range]>?>*;
```

Verilog-AMS Operators

Arithmetic	+, -, *, /, ** (exponentiation), % (modulus)
Relational	<, >, ≤, ≥, ==, !=, ===, !==
Logical	! (not), && (and), || (or)
Bitwise	~ (not), & (and), | (or), ^ (xor), ^~ or ~^ (xnor)
Reduction (unary)	& (and), ~& (nand), | (or), ~| (nor), ^ (xor), ^~ or ~^ (xnor)
Logical Shift	<<, >>
Arithmetic Shift	<<<, >>>
Ternary	a ? b : c (evaluates to b if a is true, otherwise c)
Concatenate	{ } (concatenate elements)
Assignment pattern	'{ } (create an array of values)
Replicate	{ n{ } } (strings) '{ n{ } } (arrays)

- Not available in Verilog-A.

Verilog-A

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

- SPICE is focused on simulating the behavior of analog circuits at the level of resistors, capacitors, and transistors. In contrast, Verilog-A allows **modeling systems at a wide range of abstraction**.

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

- SPICE is focused on simulating the behavior of analog circuits at the level of resistors, capacitors, and transistors. In contrast, Verilog-A allows **modeling systems at a wide range of abstraction**.
- SPICE is not a standardized language, which means that different SPICE implementations can have slightly different syntax and features. This can make it difficult to share and reuse SPICE models. Verilog-A, on the other hand, is a **standardized language**.

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

- SPICE is focused on simulating the behavior of analog circuits at the level of resistors, capacitors, and transistors. In contrast, Verilog-A allows **modeling systems at a wide range of abstraction**.
- SPICE is not a standardized language, which means that different SPICE implementations can have slightly different syntax and features. This can make it difficult to share and reuse SPICE models. Verilog-A, on the other hand, is a **standardized language**.
- Verilog-A is **more readable and easier to write** than SPICE. This can make it more convenient to create and maintain complex analog circuits.

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

- SPICE is focused on simulating the behavior of analog circuits at the level of resistors, capacitors, and transistors. In contrast, Verilog-A allows **modeling systems at a wide range of abstraction**.
- SPICE is not a standardized language, which means that different SPICE implementations can have slightly different syntax and features. This can make it difficult to share and reuse SPICE models. Verilog-A, on the other hand, is a **standardized language**.
- Verilog-A is **more readable and easier to write** than SPICE. This can make it more convenient to create and maintain complex analog circuits.
- SPICE does not provide support for modeling digital circuits or mixed-signal systems. **Verilog-AMS**, a superset of Verilog-A, provides support for both analog and digital modeling.

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

- SPICE is focused on simulating the behavior of analog circuits at the level of resistors, capacitors, and transistors. In contrast, Verilog-A allows **modeling systems at a wide range of abstraction**.
- SPICE is not a standardized language, which means that different SPICE implementations can have slightly different syntax and features. This can make it difficult to share and reuse SPICE models. Verilog-A, on the other hand, is a **standardized language**.
- Verilog-A is **more readable and easier to write** than SPICE. This can make it more convenient to create and maintain complex analog circuits.
- SPICE does not provide support for modeling digital circuits or mixed-signal systems. **Verilog-AMS**, a superset of Verilog-A, provides support for both analog and digital modeling.
- Verilog-A can be used to model **multidisciplinary systems**, which are systems that include multiple domains such as electrical, mechanical, and thermal.

Why use Verilog-A?

Verilog-A is a **modeling language** for analog circuits.

- SPICE is focused on simulating the behavior of analog circuits at the level of resistors, capacitors, and transistors. In contrast, Verilog-A allows **modeling systems at a wide range of abstraction**.
- SPICE is not a standardized language, which means that different SPICE implementations can have slightly different syntax and features. This can make it difficult to share and reuse SPICE models. Verilog-A, on the other hand, is a **standardized language**.
- Verilog-A is **more readable and easier to write** than SPICE. This can make it more convenient to create and maintain complex analog circuits.
- SPICE does not provide support for modeling digital circuits or mixed-signal systems. **Verilog-AMS**, a superset of Verilog-A, provides support for both analog and digital modeling.
- Verilog-A can be used to model **multidisciplinary systems**, which are systems that include multiple domains such as electrical, mechanical, and thermal.
- Verilog-A enables the creation of **simulator-agnostic** compact models, overcoming the limitations of older methods that used C or Fortran. These legacy models were tightly integrated with specific numerical algorithms, restricting their portability across different simulators.

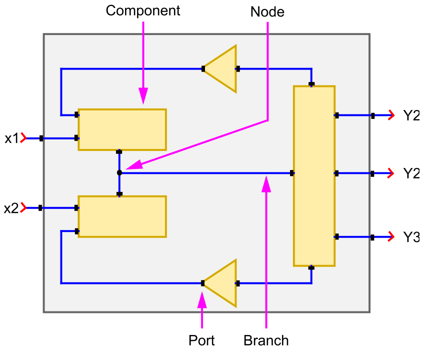
Verilog-A Module Template

```
`include "disciplines.vams" // natures and disciplines
`include "constants.vams"   // common physical and math constants

module module_name (port1, port2);
    inout port1, port2;
    electrical port1, port2;
    (* desc="Shunt resistance", units="Ohm" *) parameter real input1 = 1.0 from [0:inf];
    (* desc="Switching direction", units="" *) parameter integer input2 = 1 from [-1:1] exclude 0;
    parameter string mos_type = "NMOS" from {"NMOS", "PMOS"};
    real X;
    // this is a single line comment
    /* this is a
     * comment block */
    analog begin
        @( initial_step ) begin
            // performed at the first timestep of an analysis
        end
        if (input2 > 0) begin : local_block_name
            $strobe("input2 is positive", input1);

            // module behavioral description:
            V(port1, port2) <+ I(port1, port2) * input1;
        end
        @( final_step ) begin
            // performed at the last time step of an analysis
        end
    end
endmodule
```

Analog Modeling Concepts

Describing a System

- A **node** is a physical point where two or more circuit elements are connected together.
- A **net** is the name used for a collection of nodes that share the same electrical potential.
- A **port** is a net that traverses the boundary of a module.
- A **branch** is a path between two nets.
- A **signal** is a physical quantity that varies over time, such as voltage or current.



Structural and Behavioral Description

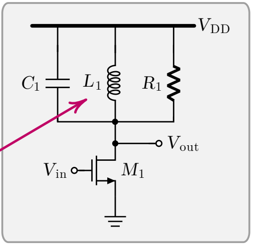
A system can be described at different levels of abstraction:

- **Behavioral Level**: At the lowest level, a system is described in terms of the mathematical relationships of its signals.
- **Structural Level**: At the highest level, a system is described as the interconnection of submodules.

Linear inductor

```
`include "disciplines.vams"

module inductor (p, n);
    parameter real L=1.0n;
    inout p, n;
    electrical p, n;

    analog
        V(p,n) <+ L * ddt(I(p,n));
endmodule
```

Behavioral



Structural

Behavioral Level Modeling

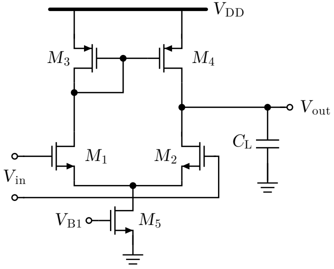
An analog circuit can be modeled in two ways:

- **Conservative System Modeling**:
  - This takes into account Kirchhoff's circuit laws, ensuring that energy is conserved.
  - Suitable for modeling systems where bidirectional interactions are important.
- **Signal-Flow Modeling**:
  - Focuses on the flow of signals from inputs to outputs.
  - Suitable for modeling systems where the signal flow is unidirectional.

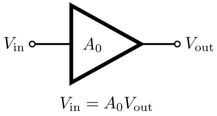
Behavioral Level Modeling

An analog circuit can be modeled in two ways:

- **Conservative System Modeling**:
  - This takes into account Kirchhoff's circuit laws, ensuring that energy is conserved.
  - Suitable for modeling systems where bidirectional interactions are important.
- **Signal-Flow Modeling**:
  - Focuses on the flow of signals from inputs to outputs.
  - Suitable for modeling systems where the signal flow is unidirectional.



Conservative



Signal-Flow

Literals and Variables

Literals

- **Numbers**: Integers and Reals.
  - Reals can be specified in decimal notation (e.g. `12_000`), scientific notation (e.g. `12e3` or `12E3`), or using scale factors (e.g. `12k`).
  - Scale factors: (10^{12}) `T`, `G`, `M`, `K`, `m`, `u`, `n`, `p`, `f`, `a` (10^{-15}).

Literals

- **Numbers**: Integers and Reals.
  - Reals can be specified in decimal notation (e.g. `12_000`), scientific notation (e.g. `12e3` or `12E3`), or using scale factors (e.g. `12k`).
  - Scale factors: (10^{12}) `T, G, M, K, m, u, n, p, f, a` (10^{-15}).
- **Strings**: They are given as sequence of characters enclosed in double quotes.
  - The concatenation operator (`{ }`) can be used to concatenate two or more strings.

    ```
    {"a", "b", "c"} // yields "abc"
    ```

  - The replication operator (`n{ }`) can be used to replicate strings.

    ```
    {"a", 2{"b", "c"}, "d"} // yields "abcbcd"
    ```

Literals

- **Numbers**: Integers and Reals.
  - Reals can be specified in decimal notation (e.g. `12_000`), scientific notation (e.g. `12e3` or `12E3`), or using scale factors (e.g. `12k`).
  - Scale factors: (10^{12}) `T, G, M, K, m, u, n, p, f, a` (10^{-15}).
- **Strings**: They are given as sequence of characters enclosed in double quotes.
  - The concatenation operator (`{ }`) can be used to concatenate two or more strings.

    ```
    {"a", "b", "c"} // yields "abc"
    ```
  - The replication operator (`n{ }`) can be used to replicate strings.

    ```
    {"a", 2{"b", "c"}, "d"} // yields "abcbcd"
    ```
- **Lists**: A list is created using the assignment pattern operator (`'{ }`), which combines its arguments into an array. The individual arguments may be scalars or lists.

```
'{4, 8, 12, 16, 20}                // a list containing scalars
'{4.0, 8.0, '{12.0, 16.0, 20.0}} // a list can also contain other lists
```

Constants

- Commonly-used **mathematical** and **physical** constants are defined in the file `constants.vams`.
- Mathematical constants are prefixed with `` `M_``, whereas physical constants use the `` `P_`` prefix.
- This file can be included using `` `include "constants.vams"``.

Mathematical Constants

```
`M_PI          // pi
`M_TWO_PI      // 2*pi
`M_PI_2        // pi/2
`M_PI_4        // pi/4
`M_1_PI        // 1/pi
`M_2_PI        // 2/pi
`M_2_SQRTPI    // 2/rt(pi)
`M_E           // e
`M_LOG2E       // log2(e)
`M_LOG10E      // log10(e)
`M_LN2         // ln(2)
`M_LN10        // ln(10)
`M_SQRT2       // rt(2)
`M_SQRT1_2     // rt(1/2)
```

Physical Constants

```
`P_Q           // charge of an electron
`P_C           // speed of light
`P_K           // Boltzmann's constant
`P_H           // Planck's constant
`P_EPS0        // permittivity of a vacuum
`P_U0          // permeability of a vacuum
`P_CELSIUS0    // 0 Celsius
```

Variables

The syntax for declaring variables is as follows:

```
<integer | real | string> identifier<[range]>? <, identifier<[range]>?>*;
range ::= integer:integer
```

Example

```
integer a[1:64];        // declares an array of 64 integers
integer b, c, d[-20:0]; // declares 2 integers and an array
```

Variables

The syntax for declaring variables is as follows:

```
<integer | real | string> identifier<[range]>? <, identifier<[range]>?>*;
range ::= integer:integer
```

Example

```
integer a[1:64];        // declares an array of 64 integers
integer b, c, d[-20:0]; // declares 2 integers and an array
```

- The subscript operator ([ ]) can be used to access the elements of a array.

Variables

The syntax for declaring variables is as follows:

```
<integer | real | string> identifier<[range]>? <, identifier<[range]>?>*;
range ::= integer:integer
```

Example

```
integer a[1:64];        // declares an array of 64 integers
integer b, c, d[-20:0]; // declares 2 integers and an array
```

- The subscript operator ([ ]) can be used to access the elements of a array.
- Variables can take a constant value at declaration time.

```
real arr[1:3] = '{1.1, 2.2, 3.3};
```

Variables

The syntax for declaring variables is as follows:

```
<integer | real | string> identifier<[range]>? <, identifier<[range]>?>*;
range ::= integer:integer
```

Example

```
integer a[1:64];         // declares an array of 64 integers
integer b, c, d[-20:0]; // declares 2 integers and an array
```

- The subscript operator ([ ]) can be used to access the elements of a array.
- Variables can take a constant value at declaration time.

  ```
  real arr[1:3] = '{1.1, 2.2, 3.3};
  ```

  - If the simulator does not support this, do the initialization inside an `initial_step` block.

    ```
    real arr[1:3];
    analog begin
        @(initial_step)
            arr[1:3] = '{1.1, 2.2, 3.3};
    end
    ```

Creating Modules

Module

A **module** encapsulates a block of hardware, defining its inputs, outputs, and functionality.

```
module module_name <( port_name <, port_name>* )>?;
    module_body
endmodule
```

Module

A **module** encapsulates a block of hardware, defining its inputs, outputs, and functionality.

```
module module_name <( port_name <, port_name>* )>?;
    module_body
endmodule
```

Example

```
`include "disciplines.vams"

module res1 (p, n);
    inout p, n;
    electrical p, n;
    parameter real r=1 from (0:inf);
    parameter real tc=1.5m from [0:3m);
    real reff;
    analog begin
        @(initial_step) begin
            reff = r*(1+tc*$temperature);
        end
        I(p, n) <+ V(p, n) / reff;
    end
endmodule
```

Ports

Ports provide a way to connect modules to other modules. Each port should be given a **direction** and a **type**.

```
<input | output | inout> <[range]>? port_name <, port_name>*;
discipline <[range]>? port_name <, port_name>*;
```

Ports

Ports provide a way to connect modules to other modules. Each port should be given a **direction** and a **type**.

```
<input | output | inout> <[range]>? port_name <, port_name>*;
discipline <[range]>? port_name <, port_name>*;
```

Example

```
module module_name (p, n);
    inout p, n;      // direction
    electrical p, n; // type
    ...
endmodule
```

- There are three directions possible: `input`, `output`, and `inout`. However, nets of signal-flow disciplines may only be bound to `input` or `output` ports.
- The type of a port is declared by giving its `discipline`. This specifies the type of signals that can be associated with the port.

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

- A `discipline` is a collection of related physical signal types within a domain. For example, the `electrical` discipline includes voltage and current natures.

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

- A `discipline` is a collection of related physical signal types within a domain. For example, the `electrical` discipline includes voltage and current natures.
- A `nature` is a definition within a discipline, such as voltage, that describes the properties of a signal (e.g., units, abstol, etc).

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

- A `discipline` is a collection of related physical signal types within a domain. For example, the `electrical` discipline includes voltage and current natures.
- A `nature` is a definition within a discipline, such as voltage, that describes the properties of a signal (e.g., units, abstol, etc).
- For **conservative systems**, a discipline definition has two natures associated with it: the `potential` nature and the `flow` nature.
  - For example, the `electrical` discipline has a `voltage` and a `current` nature.

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

- A `discipline` is a collection of related physical signal types within a domain. For example, the `electrical` discipline includes voltage and current natures.
- A `nature` is a definition within a discipline, such as voltage, that describes the properties of a signal (e.g., units, abstol, etc).
- For **conservative systems**, a discipline definition has two natures associated with it: the `potential` nature and the `flow` nature.
  - For example, the `electrical` discipline has a `voltage` and a `current` nature.
- For **signal-flow systems**, a discipline definition has only one nature associated with it: the `potential` nature or the `flow` nature.
  - For example, the `voltage` discipline has a `voltage` nature.

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

- A `discipline` is a collection of related physical signal types within a domain. For example, the `electrical` discipline includes voltage and current natures.
- A `nature` is a definition within a discipline, such as voltage, that describes the properties of a signal (e.g., units, abstol, etc).
- For **conservative systems**, a discipline definition has two natures associated with it: the `potential` nature and the `flow` nature.
  - For example, the `electrical` discipline has a `voltage` and a `current` nature.
- For **signal-flow systems**, a discipline definition has only one nature associated with it: the `potential` nature or the `flow` nature.
  - For example, the `voltage` discipline has a `voltage` nature.
- A collection of common disciplines and natures are defined in the file `disciplines.vams` that is provided with all implementations of Verilog-A. This file can be included by writing `` `include "disciplines.vams" ``.

Disciplines and Natures (I)

Verilog-A uses `discipline` and `nature` to categorize signals across various domains (e.g., electrical, mechanical, fluid).

- A `discipline` is a collection of related physical signal types within a domain. For example, the `electrical` discipline includes voltage and current natures.
- A `nature` is a definition within a discipline, such as voltage, that describes the properties of a signal (e.g., units, abstol, etc).
- For **conservative systems**, a discipline definition has two natures associated with it: the `potential` nature and the `flow` nature.
  - For example, the `electrical` discipline has a `voltage` and a `current` nature.
- For **signal-flow systems**, a discipline definition has only one nature associated with it: the `potential` nature or the `flow` nature.
  - For example, the `voltage` discipline has a `voltage` nature.
- A collection of common disciplines and natures are defined in the file `disciplines.vams` that is provided with all implementations of Verilog-A. This file can be included by writing `` `include "disciplines.vams" ``.
- You are free to create your own natures and disciplines.

Natures and Disciplines (II)

The definition of the `electrical` discipline and its associated natures is shown below.

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

Natures and Disciplines (II)

The definition of the `electrical` discipline and its associated natures is shown below.

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

```
nature Voltage
    units = "V";
    access = V;
    abstol = 1u;
endnature
```

```
nature Current
    units = "A";
    access = I;
    abstol = 1p;
endnature
```

Natures and Disciplines (II)

The definition of the `electrical` discipline and its associated natures is shown below.

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

```
nature Voltage
    units = "V";
    access = V;
    abstol = 1u;
endnature
```

```
nature Current
    units = "A";
    access = I;
    abstol = 1p;
endnature
```

- All natures must contain (at least) the following three pieces of information:
    - The `units` of the quantity.
    - The function to `access` the quantity from a node, terminal, or branch.
    - The `abstol` (absolute tolerance), which represents the largest value that can always be considered negligible.

Natures and Disciplines (II)

The definition of the `electrical` discipline and its associated natures is shown below.

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

```
nature Voltage
    units = "V";
    access = V;
    abstol = 1u;
endnature
```

```
nature Current
    units = "A";
    access = I;
    abstol = 1p;
endnature
```

- All natures must contain (at least) the following three pieces of information:
  - The `units` of the quantity.
  - The function to `access` the quantity from a node, terminal, or branch.
  - The `abstol` (absolute tolerance), which represents the largest value that can always be considered negligible.
    - The absolute tolerances on the predefined natures can be overwritten.

```
`define VOLTAGE_ABSTOL 1e-3
`include "disciplines.vams"
```

Natures and Disciplines (II)

The definition of the `electrical` discipline and its associated natures is shown below.

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

```
nature Voltage
    units = "V";
    access = V;
    abstol = 1u;
endnature
```

```
nature Current
    units = "A";
    access = I;
    abstol = 1p;
endnature
```

- All natures must contain (at least) the following three pieces of information:
  - The `units` of the quantity.
  - The function to `access` the quantity from a node, terminal, or branch.
  - The `abstol` (absolute tolerance), which represents the largest value that can always be considered negligible.
    - The absolute tolerances on the predefined natures can be overwritten.

    ```
    `define VOLTAGE_ABSTOL 1e-3
    `include "disciplines.vams"
    ```

- Optional natures include `ddt_nature` and `idt_nature`, representing the time derivative and time integral, respectively. For example, the `idt_nature` of `voltage` is `flux`.

Nets

- A **net** is the name used for a node within a module. They can be declared as scalar or vector nets.

```
discipline <[range]>? identifier <, identifier>*;
discipline <[range]>? identifier = initalizer;
```

- Nets with continuous disciplines are allowed to have initializers. The solver uses the initializer as a nodeset value for the potential of the net.

```
ground gnd;                               // declares ground net
electrical [1:10] node1;                  // declares a vector net
electrical [0:4] bus = '{2.3, 4.5, 6.0};  // declares vector net with nodeset values
```

- Nets appearing in the connection list of a module instantiation that are not accessed anywhere need not be declared.

Branches

- A **branch** is a path between two nets.
  - An **implicit** branch is referenced using its end points.
  - An **explicit** branch is referenced using its name.

```
branch (netname_1, netname_2) identifier <, identifier>*;
```

Branches

- A **branch** is a path between two nets.
    - An **implicit** branch is referenced using its end points.
    - An **explicit** branch is referenced using its name.

```
branch (netname_1, netname_2) identifier <, identifier>*;
```

- You can use the potential or flow access function to access the potential or flow of a named or unnamed branch.

V(b1)	Accesses the voltage across branch b1
V(n1)	Accesses the voltage of n1 relative to ground
V(n1, n2)	Accesses the voltage between n1 and n2
I(b1)	Accesses the current through branch b1
I(n1)	Accesses the current flowing from n1 to ground
I(n1, n2)	Accesses the current flowing from n1 to n2
I(<p1>)	Accesses the current flowing into the module through port p1

Parameters

Parameters are constants that can be modified at compile time to have values which are different from those specified in the declaration assignment. This allows customization of module instances.

Parameters

Parameters are constants that can be modified at compile time to have values which are different from those specified in the declaration assignment. This allows customization of module instances.

- The parameter declaration can contain an **optional** type specification. If not given, the parameter will take the type of the default value it is assigned.

```
parameter integer size = 16;
parameter real poles [0:1][0:2] = '{ '{0.1,0.2,0.3}, '{0.4,0.5,0.6} };
```

Parameters

Parameters are constants that can be modified at compile time to have values which are different from those specified in the declaration assignment. This allows customization of module instances.

- The parameter declaration can contain an **optional** type specification. If not given, the parameter will take the type of the default value it is assigned.

```
parameter integer size = 16;
parameter real poles [0:1][0:2] = '{ '{0.1,0.2,0.3}, '{0.4,0.5,0.6} };
```

- A parameter declaration can contain optional specifications of the **permissible range** of the values of a parameter. The keyword inf can be used to indicate infinity.

```
parameter real gain = 1 from [1:1000];
parameter real neg_rail = -15 from [-50:0);
```

Parameters

Parameters are constants that can be modified at compile time to have values which are different from those specified in the declaration assignment. This allows customization of module instances.

- The parameter declaration can contain an **optional** type specification. If not given, the parameter will take the type of the default value it is assigned.

```
parameter integer size = 16;
parameter real poles [0:1][0:2] = '{ '{0.1,0.2,0.3}, '{0.4,0.5,0.6} };
```

- A parameter declaration can contain optional specifications of the **permissible range** of the values of a parameter. The keyword inf can be used to indicate infinity.

```
parameter real gain = 1 from [1:1000];
parameter real neg_rail = -15 from [-50:0);
```

- A single value can be excluded from the possible valid values for a parameter.

```
parameter integer intval = 0 from [0:inf) exclude (10:20] exclude 5;
```

Parameters

Parameters are constants that can be modified at compile time to have values which are different from those specified in the declaration assignment. This allows customization of module instances.

- The parameter declaration can contain an **optional** type specification. If not given, the parameter will take the type of the default value it is assigned.

```
parameter integer size = 16;
parameter real poles [0:1][0:2] = '{ '{0.1,0.2,0.3}, '{0.4,0.5,0.6} };
```

- A parameter declaration can contain optional specifications of the **permissible range** of the values of a parameter. The keyword `inf` can be used to indicate infinity.

```
parameter real gain = 1 from [1:1000];
parameter real neg_rail = -15 from [-50:0);
```

- A single value can be excluded from the possible valid values for a parameter.

```
parameter integer intval = 0 from [0:inf) exclude (10:20] exclude 5;
```

Use `localparam` instead of `parameter` for parameters that should not be overidden.

Analog Behavior

Analog Block

- All behavioral statements are defined inside an **analog** block.
- There can be one analog block inside a module.
- The statements inside an analog block are executed **sequentially**.
- Some key constructs that can go inside an analog block include:
  - Sequential block statements
  - Conditional statements
  - Iterative statements
  - Assignment statements

Sequential Blocks

- A **sequential block** groups two or more statements together so that they act as a single statement.
- A sequential block begins with the begin keyword and ends with the end keyword (no semicolon).
- Adding a block identifier allows declaration of local variables for use within the block.

```
for ( j = 0 ; j < 10 ; j=j+1 ) begin
    if ( j%2 ) begin : odd  // block identifier
        integer j; // declares a local variable
        j = j+1;
        $display ("Odd numbers counted so far = %d", j);
    end else begin : even
        integer j; // declares a local variable
        j = j+1;
    end
end
```

Conditional Statements

- Use an **if** construct to run a statement under the control of specified conditions.

```
if (condition)
    statement;
else if (condition)
    statement;
else
    statement;
```

- Use a **case** construct to control which one of a series of statements runs.

```
case (select_expression)
    case_item_1:
        statement;
    case_item_2, case_item_3:
        statement;
    case_item_n:
        statement;
    default:
        statement;
endcase
```

Iterative Statements

- Use the **repeat** statement when you want a statement to run a fixed number of times.

```
repeat (count)
    statement;
```

- Use the **while** statement when you want to be able to leave a loop when an expression is no longer valid.

```
while (condition)
    statement;
```

- Use the **for** statement when you want a statement to run a fixed number of times.

```
for (initial_assignment; condition; step_assignment)
    statement;
```

- **NOTE**: `statement` must not include any of the following:
  - contribution statements (see slide 29)
  - event statements (see slide 41)
  - analog operators (see slide 51)

Assignment Statements

Three kinds of assignment statements can be made inside an analog block.

Assignment Statements

Three kinds of assignment statements can be made inside an analog block.

- The **procedural assignment operator** (`=`) assigns expressions to variables.

Assignment Statements

Three kinds of assignment statements can be made inside an analog block.

- The **procedural assignment operator** (`=`) assigns expressions to variables.
- The **contribution operator** (`<+`) assigns an expression to a signal. The assigned expression can be linear, non-linear, algebraic and/or differential functions of the input signals.
  - If there are multiple contributions to the same branch, the contributions accumulate.

```
// current sources in parallel
I(n,p) <+ expression_1;
I(n,p) <+ expression_2;
```

```
// voltage sources in series
V(n,p) <+ expression_1;
V(n,p) <+ expression_2;
```

  - The value of the target may be expressed in terms of itself.

```
I(diode) <+ IS*(limexp( V(diode) / $vt - r * I(diode) ) - 1);
```

  - It is at the end of the evaluation of the analog block that the simulator updates the signal.

Assignment Statements

Three kinds of assignment statements can be made inside an analog block.

- The **procedural assignment operator** (`=`) assigns expressions to variables.
- The **contribution operator** (`<+`) assigns an expression to a signal. The assigned expression can be linear, non-linear, algebraic and/or differential functions of the input signals.
  - If there are multiple contributions to the same branch, the contributions accumulate.

```
// current sources in parallel
I(n,p) <+ expression_1;
I(n,p) <+ expression_2;
```

```
// voltage sources in series
V(n,p) <+ expression_1;
V(n,p) <+ expression_2;
```

  - The value of the target may be expressed in terms of itself.

```
I(diode) <+ IS*(limexp( V(diode) / $vt - r * I(diode) ) - 1);
```

  - It is at the end of the evaluation of the analog block that the simulator updates the signal.
- The **indirect branch contribution operator** (`:`) is used for difficult to express mathematical statements. For example, `V(out) : V(in) == 0;` reads as "find V(out) such that V(in) is zero".

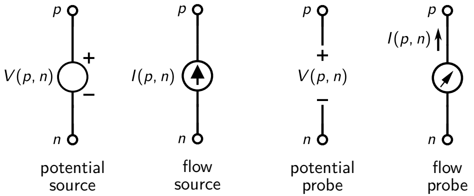
Probes and Sources

The contribution assignment statement has the form: `left_hand_side <+ right_hand_side`

Probes and Sources

The contribution assignment statement has the form: `left_hand_side <+ right_hand_side`

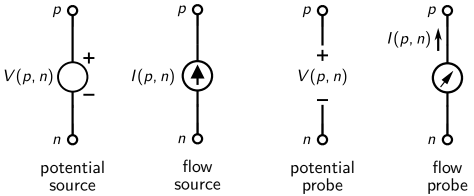
- `left_hand_side` is always a voltage or a current signal.
  - If it is a voltage signal `V(p, n)`, then the branch `(p, n)` is a **potential source**.
  - If it is a current signal `I(p, n)`, then the branch `(p, n)` is a **flow source**.
  - When we are not contributing to a branch, the branch is an open circuit. Internally, it is represented as a flow source with `I(p, n) = 0`.



Probes and Sources

The contribution assignment statement has the form: `left_hand_side <+ right_hand_side`

- `left_hand_side` is always a voltage or a current signal.
    - If it is a voltage signal `V(p, n)`, then the branch `(p, n)` is a **potential source**.
    - If it is a current signal `I(p, n)`, then the branch `(p, n)` is a **flow source**.
    - When we are not contributing to a branch, the branch is an open circuit. Internally, it is represented as a flow source with `I(p, n) = 0`.
- `right_hand_side` is an expression that may or may not include a term that depends on a signal.
    - If it includes a voltage term `V(p, n)`, then the branch `(p, n)` is a **potential probe**.
    - If it is a current term `I(p, n)`, then the branch `(p, n)` is a **flow probe**.



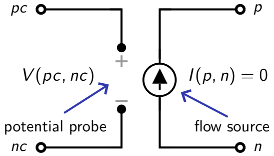
Example: Switch Branches

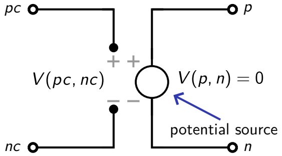
- An ideal relay has two states: “open” and “closed”.
    - For a voltage-controlled relay, exceeding the threshold voltage switches the relay from open to closed.
    - When the controlling voltage drops below the threshold, the relay switches from closed to open.

Example: Switch Branches

- An ideal relay has two states: “open” and “closed”.
    - For a voltage-controlled relay, exceeding the threshold voltage switches the relay from open to closed.
    - When the controlling voltage drops below the threshold, the relay switches from closed to open.

```
module relay (pc, nc, p, n);
    inout pc, nc, p, n;
    electrical pc, nc, p, n;
    parameter real thres=0.0;
    analog begin
        if (V(pc, nc) > thresh)
            V(p, n) <+ 0; // potential source
        else              // the else block could be omitted
            I(p, n) <+ 0; // flow source
    end
endmodule
```





Instantiating Modules

Module Instantiation

- The general syntax module instantiation is as follows:

```
module_name <#(param_list)>? instance_name (<signal_list>?);
param_list  ::= .param_name(value) <, .param_name(value)>*
signal_list ::= port_name(signal) <, port_name(signal)>*
```

Module Instantiation

- The general syntax module instantiation is as follows:

```
module_name <#(param_list)>? instance_name (<signal_list>?);
param_list  ::= .param_name(value) <, .param_name(value)>*
signal_list ::= port_name(signal) <, port_name(signal)>*
```

- The parameter list or the signal list can be given as an **unordered** comma separated list of name-value pairs, or an **ordered** comma separated list of values without names.

Assume a module has two parameters (or ports) `p1` and `p2` that were declared in that order.

- Suppose we want to assign values `v1` and `v2` to parameters (or ports) `p1` and `p2`.
  - `#(v1, v2)` (ordered)
  - `#(.p1(v1), .p2(v2))` (unordered)
- Suppose we want to assign a value only to `p2`.
  - `#(, v2)`
  - `#(.p1(), .p2(v2))`

Module Instantiation Example

```
`include "disciplines.vams"
module resistor (p, n);
    parameter real R=0;
    inout p, n;
    electrical p, n;
    analog
        V(p,n) <+ R * I(p,n);
endmodule
```

```
`include "disciplines.vams"
module vsrc (p, n);
    parameter real dc=0;
    output p, n;
    electrical p, n;
    analog
        V(p,n) <+ dc;
endmodule
```

Module Instantiation Example

```
`include "disciplines.vams"
module resistor (p, n);
    parameter real R=0;
    inout p, n;
    electrical p, n;
    analog
        V(p,n) <+ R * I(p,n);
endmodule
```

```
`include "disciplines.vams"
module vsrc (p, n);
    parameter real dc=0;
    output p, n;
    electrical p, n;
    analog
        V(p,n) <+ dc;
endmodule
```

Example: A voltage source connected to two resistors in series is shown below.

```
`include "disciplines.vams"
`include  "vsrc.va"
`include  "resistor.va"
module vdb;

    electrical n1, n2; // not accessed
    ground gnd;

    vsrc     #(.dc(1)) V1(n1, gnd);
    resistor #(.R(1k)) R1(n2, n1);
    resistor #(.R(1k)) R2(n2, gnd);
endmodule
```

Hierarchical Names

- Each signal/node/parameter identifier in a module has a unique hierarchical path name.
- The hierarchical path name uses the period (.) period to separate the names in the hierarchy.

```
`include "disciplines.vams"
`include  "vsrc.va"
`include  "resistor.va"
module vdb;
    electrical n1, n2; // not accessed
    ground gnd;

    vsrc     #(.dc(1)) V1(n1, gnd);
    resistor #(.R(1k)) R1(n2, n1);
    resistor #(.R(1k)) R2(n2, gnd);
endmodule
```

- In the example above, we can access the ports and parameter values of resistor R1 inside the vdb module as follows:

```
R1.p // access ports
R1.n
R1.R // access parameters
```

defparam

There are two other way parameter values can be overridden: `defparam` and `paramset`.

defparam

There are two other way parameter values can be overridden: `defparam` and `paramset`.

- Using the `defparam` statement, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter.

```
module resistor (p, n);
    ...
    parameter real R = 1.0;
    ...
endmodule
```

```
module top;
    ...
    resistor R1 (net1, net2);
    resistor R2 (net2, net3);
    ...

defparam
    R1.R = 10.0,
    R2.R = 20.0;
endmodule
```

defparam

There are two other way parameter values can be overridden: `defparam` and `paramset`.

- Using the `defparam` statement, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter.

```
module resistor (p, n);
    ...
    parameter real R = 1.0;
    ...
endmodule
```

```
module top;
    ...
    resistor R1 (net1, net2);
    resistor R2 (net2, net3);
    ...

defparam
    R1.R = 10.0,
    R2.R = 20.0;
endmodule
```

- Precedence rule for parameter overriding:
    - If overrides take place at different levels of the module hierarchy, the highest level override takes precedence.
    - Parameter overrides done by `defparam` take precedence over parameter overrides done by module instantiation statements.

paramset

- Use the paramset declaration to declare a set of parameters for a particular module, such that each instance of the paramset need only provide overrides for a smaller number of parameters.
- Multiple paramsets can be declared using the same paramset name. During elaboration, the simulator shall choose the appropriate paramset.

```
module baseModule (in, out);
    parameter real a = 0;
    parameter real b = 0;
    parameter real c = 0;
endmodule

paramset ps baseModule;
    parameter real a = 1.0 from [0:1];
    parameter real b = 1.0 from [0:1];
    .a = a; .b = b;
endparamset

paramset ps baseModule;
    parameter real b = 2.0 from (1:2];
    parameter real c = 1.0 from [0:1];
    .b = b; .c = c;
endparamset
```

```
// instantiation
ps #(.b(1.5)) inst1 (in, out); // the second paramset will be chosen
```

## mfactor

Circuit designers use $mfactor to mimic parallel copies of identical devices without having to instantiate large sets of devices in parallel.

- The $mfactor parameter is implicitly declared for every module.
- The $mfactor of a module can be changed like any other parameter. For example #(.$mfactor(3)) changes the $mfactor of that module to 3.0 (default is 1.0).
- All contributions to a branch flow quantity in the analog block are **automatically** multiplied by $mfactor.

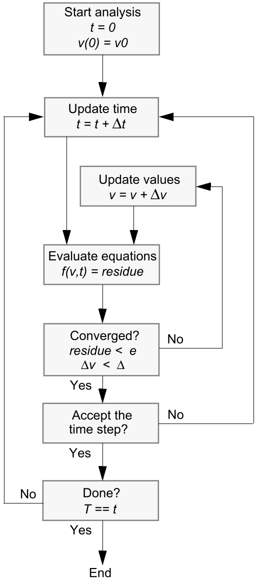
```
module badres (a, b);
    inout a, b;
    electrical a, b;
    parameter real r = 1.0 from (0:inf);
    analog begin
        I(a,b) <+ V(a,b) / r * $mfactor; // current will be multiplied by $mfactor twice
    end
endmodule
```

- The value of the inherited $mfactor in a particular module instance is the product of the $mfactor values in the ancestors of the instance and of the $mfactor value in the instance itself.

Analog Events

Simulator Flow

- Every analog simulator has the following key components:
  - Netlist parser
  - Model libraries
  - Numerical Solver
- The simulation process includes the following stages:
  - **Initialization**: Set initial conditions and prepare the circuit for simulation.
  - **Time-Step Calculation**: Determine appropriate time steps for accurate simulation.
  - **Iteration**: At each time step, solve the differential equations to update circuit state.
  - **Convergence Check**: Ensure the solution converges to a stable state within specified tolerances.
  - **Event Detection**: Identify and handle events that occur during the simulation (e.g., zero crossings).



Analog Events

- During simulation, the simulator generates events (`initial_step` and `final_step`) automatically for initialization and cleanup.
- The simulator can also be made to generate other events based on specific criteria (`cross`, `above`, `timer`).
- Verilog-A uses the `@` block construct to execute a block of code when an event occurs.

```
@(event_expression)
    <action_statement>?;
```

- The event expression generates events at particular instants of time and forces the simulator to evaluate the block of code only at those events.
- If `action_statement` is a blank statement, even though no action will be performed, it will force the simulator to place an evaluation point at or very close to the event.
- If you want to detect more than one kind of event, you can use the `or` operator.

```
@(event_expression_1 or event_expression_2)
    <action_statement>?;
```

Initial Step and Final Step

Initial Step

- The simulator generates an initial_step event during the solution of the first point in specified analyses, or, if no analyses are specified, during the solution of the first point of every analysis.
- Use the initial_step event to perform an action that should occur only at the beginning of an analysis.

```
@(initial_step<(analysis_string <, analysis_string>*)>?)
```

Initial Step and Final Step

Initial Step

- The simulator generates an `initial_step` event during the solution of the first point in specified analyses, or, if no analyses are specified, during the solution of the first point of every analysis.
- Use the `initial_step` event to perform an action that should occur only at the beginning of an analysis.

```
@(initial_step<(analysis_string <, analysis_string>*)>?)
```

Final Step

- The simulator generates a `final_step` event during the solution of the last point in specified analyses, or, if no analyses are specified, during the solution of the last point of every analysis.
- Use the `final_step` event to perform an action that should occur only at the end of an analysis.

```
@(final_step<(analysis_string <, analysis_string>*)>?)
```

Initial Step and Final Step

Initial Step

- The simulator generates an `initial_step` event during the solution of the first point in specified analyses, or, if no analyses are specified, during the solution of the first point of every analysis.
- Use the `initial_step` event to perform an action that should occur only at the beginning of an analysis.

```
@(initial_step<(analysis_string <, analysis_string>*)>?)
```

Final Step

- The simulator generates a `final_step` event during the solution of the last point in specified analyses, or, if no analyses are specified, during the solution of the last point of every analysis.
- Use the `final_step` event to perform an action that should occur only at the end of an analysis.

```
@(final_step<(analysis_string <, analysis_string>*)>?)
```

If the current analysis name matches any of the analyses listed inside the parenthesis, an event is triggered.

Analysis Types

- Verilog-A simulators typically support the following analysis types:

`"dc"`	SPICE .OP (operating point analysis) or .DC (DC sweep analysis)
`"tran"`	SPICE .TRAN (transient analysis)
`"ac"`	SPICE .AC (small-signal analysis)
`"noise"`	SPICE .NOISE (noise analysis)
`"ic"`	SPICE .IC (initial condition analysis) which precedes a transient analysis

Analysis Types

- Verilog-A simulators typically support the following analysis types:

`"dc"`	SPICE .OP (operating point analysis) or .DC (DC sweep analysis)
`"tran"`	SPICE .TRAN (transient analysis)
`"ac"`	SPICE .AC (small-signal analysis)
`"noise"`	SPICE .NOISE (noise analysis)
`"ic"`	SPICE .IC (initial condition analysis) which precedes a transient analysis

- The `analysis` function provides a way to test the current analysis and have the module behave differently depending on which analysis is being run. It returns true (1) if any argument matches the current analysis type, or false (0) if no matches are found.

```
if (analysis("ic")) // analysis("ic", "dc") to test for more than one analysis
    V(cap) <+ initial_value;
else
    I(cap) <+ C * ddt(V(cap));
```

Cross Event Function

The cross function generates events when the signal crosses zero in the specified direction dir. For any other transitions of the signal, the function does not generate an event.

```
@(cross(signal<, dir<, time_tol<, expr_tol>?>?>?)
```

Cross Event Function

The `cross` function generates events when the `signal` crosses zero in the specified direction `dir`. For any other transitions of the signal, the function does not generate an event.

```
@(cross(signal<, dir<, time_tol<, expr_tol>?>?>?)
```

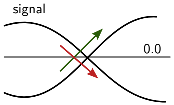


Figure: `dir=0` (default)

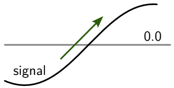


Figure: `dir=+1`

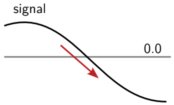


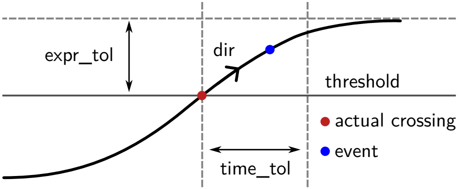
Figure: `dir=-1`

Cross Event Function

The `cross` function generates events when the `signal` crosses zero in the specified direction `dir`. For any other transitions of the signal, the function does not generate an event.

```
@(cross(signal<, dir<, time_tol<, expr_tol>?>?>?)
```

- The function coordinates with the simulator to ensure that the simulator places an evaluation point very near the threshold crossing.
- The `expr_tol` and `time_tol` arguments are absolute tolerances that represent the maximum allowable error between the true crossing point and when the cross event actually triggers.



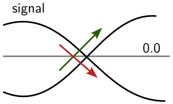


Figure: `dir=0` (default)

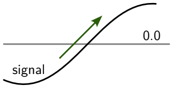


Figure: `dir=+1`

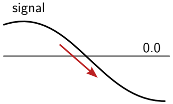
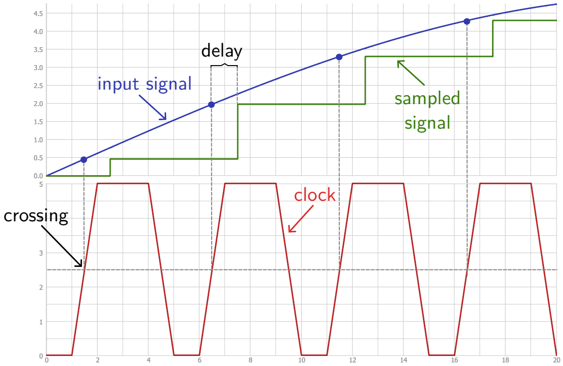


Figure: `dir=-1`

Example: Sample and Hold

```
analog begin
    @(cross(V(clk) - 2.5, +1.0))
        state = V(in);
    V(out) <+ transition(state, 1m, 0.1u);
end
```



Above Event Function

The `above` function is similar to the `cross` function, except for two things:

- It generates events only on rising transitions.
- Unlike `cross`, which fires after at least one transient time step is complete, `above` will fire at time $t = 0$. Therefore, it can be used in both DC and transient analysis.
- After time $t = 0$, the `above` function then behaves the same as a `cross` function with `dir = +1`.

Example: Digital Inverter

A naive implementation of a digital inverter is given below:

```
analog begin
    @(cross(V(in) - vth))
        lv = V(in) > vth;
    V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

Example: Digital Inverter

A naive implementation of a digital inverter is given below:

```
analog begin
    @(cross(V(in) - vth))
        lv = V(in) > vth;
    V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

- Because the `cross` function does not trigger during a DC analysis, the inverter will not know the value of its input at $t = 0$.

Two possible ways to solve this are given below.

Example: Digital Inverter

A naive implementation of a digital inverter is given below:

```
analog begin
    @(cross(V(in) - vth))
        lv = V(in) > vth;
    V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

- Because the `cross` function does not trigger during a DC analysis, the inverter will not know the value of its input at $t = 0$.

Two possible ways to solve this are given below.

```
analog begin
   @(above(V(in) - vth) or above(vth - V(in)))
      lv = V(in) > vth;
   V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

Example: Digital Inverter

A naive implementation of a digital inverter is given below:

```
analog begin
    @(cross(V(in) - vth))
        lv = V(in) > vth;
    V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

- Because the `cross` function does not trigger during a DC analysis, the inverter will not know the value of its input at $t = 0$.

Two possible ways to solve this are given below.

```
analog begin
   @(above(V(in) - vth) or above(vth - V(in)))
      lv = V(in) > vth;
   V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

```
analog begin
   @(initial_step or cross(V(in) - vth))
      lv = V(in) > vth;
   V(out) <+ transition(lv ? vh : vl, 0, tt);
end
```

Last Crossing

- The last_crossing function returns the simulation time when a signal last crossed zero in the specified direction.

```
last_crossing(signal, dir)
```

- Unlike cross and above, the last_crossing function does not control the timestep to get accurate results; it uses interpolation to estimate the time of the last crossing. However, it can be used with the cross or above function for improved accuracy.

```
t0 = last_crossing(V(start) - thresh, dir);
@(cross(V(start) - thresh, dir))
    statement;
```

Timer Event Function

- The timer function schedules an event that occurs at, or just beyond start_time.
- If the period is specified and is greater than zero, the timer function schedules subsequent events at multiples of period from start_time.

```
@(timer(start_time<, period>?)
```

Timer Event Function

- The timer function schedules an event that occurs at, or just beyond start_time.
- If the period is specified and is greater than zero, the timer function schedules subsequent events at multiples of period from start_time.

```
@(timer(start_time<, period>?)
```

An example of a square wave generator is given below.

```
module squarewave (out);
    output out;
    electrical out;
    parameter real period = 1.0;
    integer x;

    analog begin
        @(initial_step)
            x = 1;
        @(timer(0, period/2))
            x = !x;
        V(out) <+ transition(x, 0.0, period/100.0);
    end
endmodule
```

Analog Operators

Analog Operators

- Analog operators are functions that operate on more than just the current value of their arguments. These functions maintain an internal state and produce a return value that is a function of an input expression, the arguments, and their internal state.
- Because they maintain their internal state, analog operators are subject to several important **restrictions** to prevent the internal state from becoming corrupted or out-of-date.
  - They can only be used inside an `analog` process.
  - You can use analog operators inside an `if` or `case` construct only if the controlling conditional expression consists entirely of `genvar` expressions, literal numerical constants, parameters, or the analysis function.
  - You cannot use analog operators in `repeat` or `while` statements.
  - You can only use them inside `for` statements if the initialization index of type `genvar`.

Using Analog Operators Inside For Loops

In order to use analog operators within a **for** statement, Verilog-A requires the loop index be of type `genvar`.

- A `genvar` variable can consist only of expressions of static values: the expressions can consist of operations on parameters, literals, and other genvar variables.
- These restrictions result in the bounds of the loop being static. As such, they are known before the simulation begins and they cannot change during the simulation.

Using Analog Operators Inside For Loops

In order to use analog operators within a **for** statement, Verilog-A requires the loop index be of type `genvar`.

- A `genvar` variable can consist only of expressions of static values: the expressions can consist of operations on parameters, literals, and other genvar variables.
- These restrictions result in the bounds of the loop being static. As such, they are known before the simulation begins and they cannot change during the simulation.

```
genvar i;
analog begin
    ...
    for (i=0; i<N; i=i+1)
        V(out[i]) <+ transition(result[i], td, tt);
    ...
end
```

Commonly-used Analog Operators

- Use ddt to calculate the time derivative of a signal.

```
ddt(signal)
```

Commonly-used Analog Operators

- Use ddt to calculate the time derivative of a signal.

```
ddt(signal)
```

- Use idt to calculate the time integral of a signal.

```
idt(signal, ic) // ic is the initial condition
```

Commonly-used Analog Operators

- Use ddt to calculate the time derivative of a signal.

```
ddt(signal)
```

- Use idt to calculate the time integral of a signal.

```
idt(signal, ic) // ic is the initial condition
```

- Use idtmod when you want the output to wrap so that it always falls between offset and offset + modulus (circular integration).

```
idtmod(signal, ic, modulus, offset)
```

Commonly-used Analog Operators

- Use ddt to calculate the time derivative of a signal.

```
ddt(signal)
```

- Use idt to calculate the time integral of a signal.

```
idt(signal, ic) // ic is the initial condition
```

- Use idtmod when you want the output to wrap so that it always falls between `offset` and `offset + modulus` (circular integration).

```
idtmod(signal, ic, modulus, offset)
```

- Use transition to convert a piecewise constant waveform into a waveform that has controlled transitions. It is mostly used on discrete-valued signals to avoid sudden jumps.

```
transition(signal, delay, t_rise, t_fall)
```

Commonly-used Analog Operators

- Use ddt to calculate the time derivative of a signal.

```
ddt(signal)
```

- Use idt to calculate the time integral of a signal.

```
idt(signal, ic) // ic is the initial condition
```

- Use idtmod when you want the output to wrap so that it always falls between offset and offset + modulus (circular integration).

```
idtmod(signal, ic, modulus, offset)
```

- Use transition to convert a piecewise constant waveform into a waveform that has controlled transitions. It is mostly used on discrete-valued signals to avoid sudden jumps.

```
transition(signal, delay, t_rise, t_fall)
```

- Use slew to bound the slope of a continuous signal.

```
slew(signal, max_positive_slope, max_negitive_slope)
```

Laplace Transform S-Domain Filters

The Laplace Transform (LT) operators implement continuous-time filters. There are four forms of the LT operator, depending on how the numerator and denominator of the transfer function are expressed.

```
laplace_nd(input, numer, denom) // polynomial / polynomial
laplace_np(input, numer, denom) // polynomial / poles
laplace_zd(input, numer, denom) // zeros / polynomial
laplace_zp(input, numer, denom) // zeros / poles
```

Laplace Transform S-Domain Filters

The Laplace Transform (LT) operators implement continuous-time filters. There are four forms of the LT operator, depending on how the numerator and denominator of the transfer function are expressed.

```
laplace_nd(input, numer, denom) // polynomial / polynomial
laplace_np(input, numer, denom) // polynomial / poles
laplace_zd(input, numer, denom) // zeros / polynomial
laplace_zp(input, numer, denom) // zeros / poles
```

Consider the transfer function:

$$H(s) = \frac{N(s)}{D(s)} = \frac{1}{(s+1)(s+2)} = \frac{1}{s^2+3s+2}$$

The filter can be expressed in one of the following forms:

```
laplace_nd(input, {1.0}, {2.0, 3.0, 1.0})
laplace_np(input, {1.0}, {-1.0, 0.0, -2.0, 0.0})
laplace_zd(input, {1.0}, {2.0, 3.0, 1.0})
laplace_zp(input, {1.0}, {-1.0, 0.0, -2.0, 0.0})
```

System Functions

System Functions

- All system functions start with the $ symbol.

System Functions

- All system functions start with the $ symbol.
- The following system functions return information about the current circuit environment in the form of a real number.

`$abstime`	returns the current time in seconds
`$realtime`	returns the current time in time units
`$temperature`	returns the current ambient temperature in Kelvin
`$vt`	returns the thermal voltage corresponding to the current temperature

System Functions

- All system functions start with the $ symbol.
- The following system functions return information about the current circuit environment in the form of a real number.

`$abstime`	returns the current time in seconds
`$realtime`	returns the current time in time units
`$temperature`	returns the current ambient temperature in Kelvin
`$vt`	returns the thermal voltage corresponding to the current temperature

- The following system functions allow the model to control simulation flow in order to help the solver maintain accurate results in exceptional situations.
  - `$bound_step(max_step)` controls the maximum time step the simulator takes during a transient simulation.
  - `$discontinuity(order)` provides the simulator information about known discontinuities to provide help for simulator convergence algorithms. The argument indicates the lowest order derivative that is discontinuous.
    - If there's discontinuity in the waveform, then set `order` to 0.
    - If there's discontinuity in the slope of the waveform, then set `order` to 1.

I/O Functions

Displaying Outputs

- Verilog-A provides these tasks for displaying information during a simulation: `$strobe`, `$display`, `$write`, `$debug`, and `$monitor`.
  - `$strobe` prints a newline character after the final argument, whereas `$display` does not.

Displaying Outputs

- Verilog-A provides these tasks for displaying information during a simulation: `$strobe`, `$display`, `$write`, `$debug`, and `$monitor`.
  - `$strobe` prints a newline character after the final argument, whereas `$display` does not.
- The functions have the following form:

```
$function_name(<format_string>? argument <, argument>*)
```

Displaying Outputs

- Verilog-A provides these tasks for displaying information during a simulation: `$strobe`, `$display`, `$write`, `$debug`, and `$monitor`.
  - `$strobe` prints a newline character after the final argument, whereas `$display` does not.
- The functions have the following form:

```
$function_name(<format_string>? argument <, argument>*)
```

  - `format_string` is a string that can include strings with a special syntax.

```
% <flag>? <field_width>? <. precision>? format_character
```

Displaying Outputs

- Verilog-A provides these tasks for displaying information during a simulation: `$strobe`, `$display`, `$write`, `$debug`, and `$monitor`.
  - `$strobe` prints a newline character after the final argument, whereas `$display` does not.
- The functions have the following form:

```
$function_name(<format_string>? argument <, argument>*)
```

  - `format_string` is a string that can include strings with a special syntax.

```
% <flag>? <field_width>? <. precision>? format_character
```

    - `flag` can be − (left justify the output) or + (always print a sign).
    - `field_width` is an integer specifying the minimum width for the field.
    - `precision` is an integer specifying the number of digits to the right of the decimal point
    - Common format characters include `d` (integer) `e` (real, exponential format), `f` (real, fixed-decimal format), `g` (real, exponential or decimal format, whichever is shorter), and `s` (string).

Opening and Closing Files

- `$fopen` opens a file by assigning it to one of 32 available output chanels.

```
channel_descriptor = $fopen(file_name <, <"r" | "w" | "a"> >?);
```

  - The `channel_descriptor` returned by `$fopen` is an unsigned integer that is uniquely associated with `file_name`. `$fopen` returns zero (0) if the file cannot be opened.
  - By default, a file is opened for writing.

Opening and Closing Files

- `$fopen` opens a file by assigning it to one of 32 available output chanels.

```
channel_descriptor = $fopen(file_name <, <"r" | "w" | "a"> >?);
```

  - The `channel_descriptor` returned by `$fopen` is an unsigned integer that is uniquely associated with `file_name`. `$fopen` returns zero (0) if the file cannot be opened.
  - By default, a file is opened for writing.

- `$fclose` closes the file specified by the `channel_descriptor`.

Opening and Closing Files

- $fopen opens a file by assigning it to one of 32 available output chanels.

```
channel_descriptor = $fopen(file_name <, <"r" | "w" | "a"> >?);
```

  - The channel_descriptor returned by $fopen is an unsigned integer that is uniquely associated with file_name. $fopen returns zero (0) if the file cannot be opened.
  - By default, a file is opened for writing.

- $fclose closes the file specified by the channel_descriptor.

```
analog begin
    @ (initial_step) begin
        out_file1 = $fopen("some_name.dat");
        // `%C` = file name, `:r` = base name, `:e` = extension
        out_file2 = $fopen("%C:r.dat");
    end
    // ...
    @ (final_step) begin
        $fclose(out_file_1);
        $fclose(out_file_2);
    end
end
```

Reading from a File

- `$fscanf` retrieves data from a file and returns the number of objects read.

```
num_objects = $fscanf(channel_descriptor, format_string, arg_list);
```

Reading from a File

- $fscanf retrieves data from a file and returns the number of objects read.

```
num_objects = $fscanf(channel_descriptor, format_string, arg_list);
```

- $fgets reads the characters until a newline character or EOF condition is encountered.

```
$fgets(string, channel_descriptor);
```

Reading from a File

- `$fscanf` retrieves data from a file and returns the number of objects read.

```
num_objects = $fscanf(channel_descriptor, format_string, arg_list);
```

- `$fgets` reads the characters until a newline character or EOF condition is encountered.

```
$fgets(string, channel_descriptor);
```

```
integer cnt, num_objects, mcd, intd[100:0];
real rdb1[100:0], rdb2[100:0];
analog begin
    @(initial_step) begin
        cnt = 0;
        mcd = $fopen("input.dat", "r");
    end
    while (cnt < 100) begin
        num_objects = $fscanf(mcd, "%d %e %e", intd[cnt], rdb1[cnt], rdb2[cnt]);
        cnt = cnt + 1;
    end
    @(final_step)
        $fclose(mcd);
end
```

Writing to a File

The functions that are used for writing to a file are the same as those used for printing to the screen, except that they take a channel descriptor as an extra argument.

```
$fstrobe(channel_descriptor, format_string, arg_list);
```

Writing to a File

The functions that are used for writing to a file are the same as those used for printing to the screen, except that they take a channel descriptor as an extra argument.

```
$fstrobe(channel_descriptor, format_string, arg_list);
```

```
integer mcd;

analog begin
    @(initial_step)
        mcd = $fopen("zero-crossings.dat");

    last = last_crossing(V(in), dir);
    @(cross(V(in), dir))
        $fstrobe(mcd, "%0.1e", last);

    @(final_step)
        $fclose(mcd);
end
```

User-defined Functions

Defining a Function

```
analog function <real | integer> function_name;
     function_item_declarations
     function_block
endfunction;
```

- `function_item_declaration` defines the input and output arguments to the function, as well as local variables used in the function.
    - Every function should have at least one `input` declared.
    - `output` and `inout` (**not passed by reference**) variables can be assigned to within the `statement_block`; the last value assigned during function evaluation is the value returned by the function.
    - The analog function implicitly declares a variable of the same name as the function. This variable can be assigned to within the `statement_block`; its last assigned value is returned by the function.
    - The type of the return value is optional. If unspecified, the default is `real`.

Defining a Function

```
analog function <real | integer> function_name;
     function_item_declarations
     function_block
endfunction;
```

- `function_item_declaration` defines the input and output arguments to the function, as well as local variables used in the function.
  - Every function should have at least one `input` declared.
  - `output` and `inout` (**not passed by reference**) variables can be assigned to within the `statement_block`; the last value assigned during function evaluation is the value returned by the function.
  - The analog function implicitly declares a variable of the same name as the function. This variable can be assigned to within the `statement_block`; its last assigned value is returned by the function.
  - The type of the return value is optional. If unspecified, the default is `real`.
- The `function_block` **should not** include the following: analog operators, access function, analog filter functions, contribution statements, and event control statements.

Calling a Function

To call a user-defined analog function, use the following syntax.

```
function_name(expression <, expression>*)
```

Calling a Function

To call a user-defined analog function, use the following syntax.

```
function_name(expression <, expression>*)
```

The following example defines a user-defined function called `geomcalc`, which returns both the `area` and `perimeter` of a rectangle.

```
real a, b, c, d;

analog function real geomcalc;
    input l, w;
    output area, perim;
    real l, w, area, perim;
    begin
        area = l * w;
        perim = 2 * ( l + w );
    end
endfunction

analog begin
    // ...
    geomcalc(a, b, c, d);
    // ...
end
```

Controlling the Compiler

Compiler Directives

The backtick character (`) introduces a compiler directive, which comes into effect during the compilation process. Some examples are given below.

- `` `include "disciplines.vams" ``: includes the content of the file.
- `` `define CAP(A,D) (A*EPS0)/D ``: `CAP(A,D)` can be used in place of `(A*EPS0)/D` in the code. A more elaborate example is shown below.

```
`define my_resistor(b_r, r) \
    if (r > 1.0e-3) begin \
        I(b_r) <+ V(b_r) / (r); \
        I(b_r) <+ white_noise(4.0 * `KB_NIST2004 * $temperature/(r)); \
    end else begin \
        V(b_r) <+ I(b_r) * (r); \
        V(b_r) <+ white_noise(4.0 * `KB_NIST2004 * $temperature*(r)); \
    end
```

- `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `endif ``: marks sections of code that can be conditionally ignored.

```
`define DEBUG 0

// somehwere else in the code
`ifdef DEBUG
    $strobe("Gap: %e", gap);
`endif
```

Verilog-A Best Practices

Best Practices (I)

- Derived physical constants can change over time, so explicitly define the physical constants you use in your model. Declare them to 17 or more digits of precision.
- If a model contains repetitive use of the same calculation sequence, formulate the calculation as an analog function and place it in a separate file, then `` `include `` that file in the Verilog-A module that defines the main model.
- Always write real values as such, i.e., 2.0 and not 2 without the decimal. Not only is it an explicit visual clue as to the data type, but in Verilog-A 1/2 will be interpreted as an integer divide and will return a value of zero, not 0.5
- Specify appropriate limits for parameters. These should restrict the range of usage to where the model has been verified, and avoid values that can cause numerical evaluation problems.
- Protect arguments to functions so they do not cause numerical overflows; this may be done using `limexp` for exponentials.
- Always declare branches and use only those defined branches as the argument to access functions.
- If you access a port current, for example to print as part of operating point information, remember to access it via the port access function `I(<port>)` rather than the branch access function `I(port)` as the latter effectively shorts the port to ground, which is likely not what was intended.

Best Practices (II)

- Do not use event statements. They are intended for behavioral modeling, not compact modeling.
- Ensure that each usage of a macro argument in the body of the macro has parentheses (.) around the argument.

```
`define myatan2(y,x) atan(y/x)     // wrong
`define myatan2(y,x) atan((y)/(x)) // correct
```

- Formulate models in terms of the proper discipline; even though you can, do not map non-electrical disciplines (e.g., temperature and heat flow, for thermal modeling) into voltage and current. This is because the scale of typical values for a discipline may be different from the electrical discipline, hence the electrical discipline convergence criteria may not be appropriate.
- For compact modeling, make sure your model does not inherit values from the previous iterations. These are known as hidden states. To avoid this, either (1) make sure all quantities used in all branches of a conditional are defined in all branches of a conditional block that it depends on, or (2) define an initial value for such variables outside the conditional blocks, to guarantee that there are no unintentional hidden states.

Best Practices (III)

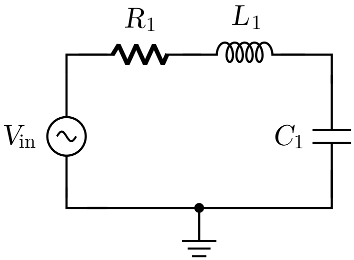
- Where possible, use current contributions (not voltage contributions) to minimize the number of added system unknowns. This is because circuit simulators use nodal analysis which is based on Kirchhoff's current law, where the state variables are the unknown voltages. Adding a voltage contribution statement such as `V(a,b)<+K;` necessitates the addition of an extra variable for the current flowing through the voltage source and the addition of an extra equation $V_a - V_b = K$.
- Avoid using variables that depend on `ddt` in conditionals, as this causes an extra branch equation to be created.

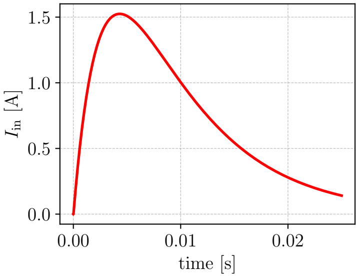
```
Qbd_ddt = ddt(Qbd);
Qbs_ddt = ddt(Qbs);
if (V(b_ds) ≥ 0.0) begin
    Ibdx_ddt = Qbd_ddt;
    Ibsx_ddt = Qbs_ddt;
end else begin
    Ibdx_ddt = Qbs_ddt;
    Ibsx_ddt = Qbd_ddt;
end
I(b_bd) <+ Ibdx_ddt;
I(b_bs) <+ Ibsx_ddt;
```

```
if (V(b_ds) ≥ 0.0) begin
    Qbdx = Qbd;
    Qbsx = Qbs;
end else begin
    Qbdx = Qbs;
    Qbsx = Qbd;
end
I(b_bd) <+ ddt(Qbdx);
I(b_bs) <+ ddt(Qbsx);
```

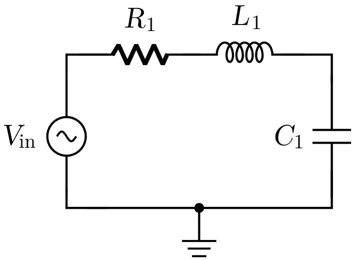
Verilog-A Simulators

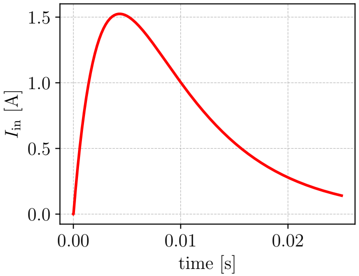
RLC Circuit





RLC Circuit





myres.va

```
`include "disciplines.vams"
module myres (p, n);
    inout p, n;
    electrical p, n;
    parameter real R = 1k;
    analog
        I(p, n) <+ V(p, n) /
        ↪  R;
endmodule
```

mycap.va

```
`include "disciplines.vams"
module mycap (p, n);
    inout p, n;
    electrical p, n;
    parameter real C = 1u;
    analog
        I(p, n) <+ C *
        ↪  ddt(V(p, n));
endmodule
```

myind.va

```
`include "disciplines.vams"
module myind (p, n);
    inout p, n;
    electrical p, n;
    parameter real L = 1u;
    analog
        V(p, n) <+ L *
        ↪  ddt(I(p, n));
endmodule
```

Spectre (Cadence)

- Save the testbench below as `rlc_tb.scs`.
- Run the following: `spectre rlc_tb.scs`.

rlc_tb.scs

```
simulator lang=spectre
global 0

I1 (n1 n2) myres R=0.5
I2 (n2 n3) mycap C=20m
I3 (n3 0) myind L=1m

Vin (n1 0) vsource type=pulse val0=0 val1=1 period=1 \
    delay=10u rise=10n fall=10n width=900.0m

save all

tran tran stop=0.025 errpreset=liberal

ahdl_include "./myres.va"
ahdl_include "./mycap.va"
ahdl_include "./myind.va"

# To save the simulation results in SPICE3 format
# settings options rawfmt=psfascii
```

Hspice (Synopsys)

- Save the testbench below as `rlc_tb.sp`.
- Run the following: `hspice -i rlc_tb.sp -o ./result/`.

rlc_tb.sp

```
*RLC transient simulation

* Save options
*.OPTION POST [= 0|1|2|3|ASCII|BINARY|CSDF]

.OPTION POST

.hdl mycap.va
.hdl myind.va
.hdl myres.va

X1 n1 n2 myres R=0.5
X2 n2 n3 mycap C=20m
X3 n3 0 myind L=1m

Vin n1 0 PULSE(0 1 10u 10n 10n 900m 1)

.tran 0.1m 0.025

.end
```

Ngpsice and OpenVAF

- Make sure that your version of ngspice is compiled with the `-enable-osdi` flag.
- Compile the Verilog-A model files: `for f in *.va; do openvaf "$f"; done`.
- Save the testbench below as `rlc_tb.sp` and run the following: `ngspice rlc_tb.sp`.

rlc_tb.sp

```
*RLC transient simulation

.model myres myres R=0.5
.model mycap mycap C=20m
.model myind myind L=1m

NX1 n1 n2 myres
NX2 n2 n3 mycap
NX3 n3 0 myind

Vin n1 0 PULSE(0 1 10u 10n 10n 900m 1)

.control
    pre_osdi ./myres.osdi ./mycap.osdi ./myind.osdi
    tran 0.1m 0.025
    plot -I(Vin)
.endc

.end
```

Verilog-A Examples

Sample and Hold

```
`include "disciplines.vams"

module sah (in, out);
    input in; voltage in;
    output out; voltage out;
    parameter real period = 1 from (0:inf);          // sampling period (s)
    parameter real toff = 0 from [0:inf);            // time before sampling (s)
    parameter real td = 0 from [0:inf);              // delay from sampling to output (s)
    parameter real tt = period/100 from [0:inf);     // output transition time (s)

    real hold;

    analog begin
        @(timer(toff, period) or initial_step) begin
            hold = V(in);
        end
        V(out) <+ transition(hold, td, tt);
    end
endmodule
```

Track and Hold

```
`include "disciplines.vams"

module tah (in, clk, out);
    input in, clk; voltage in, clk;
    output out; voltage out;
    parameter real thresh = 1 from (0:inf);     // clock threshold voltage (V)
    parameter real td = 0 from [0:inf);         // delay from sampling to output (s)
    parameter real tt = 0 from [0:inf);         // output transition time (s)

    real track;

    analog begin
        // track when the clock is HIGH
        if (V(clk) > thresh)
            track = V(in);

        // start holding when the clock goes LOW
        @(cross(V(clk)-thresh, -1))
            trach = V(in);

        V(out) <+ transition(track, td, tt);
    end
endmodule
```

Voltage Controlled Oscillator (Modeling)

A voltage-controlled oscillator (VCO) produces an output signal whose frequency is proportional to an input signal.

- A sinosoidal signal can be described as:

$$v_{\mathrm{out}}(t) = A \cdot \sin(\varphi(t)) \qquad (1)$$

where $\varphi(t)$ is the phase.

- Recall that the angular frequency ω is the derivative of the phase.

$$\omega(t) = \frac{\mathrm{d}}{\mathrm{d}t}(\varphi(t))$$

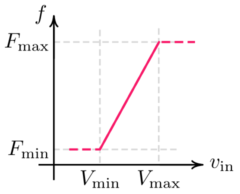
$$\varphi(t) = 2\pi \int_{-\infty}^{t} f(t)\mathrm{d}t \qquad (2)$$

- By the requirement of the VCO:

$$f(t) = f_0 + K \cdot v_{\mathrm{in}}(t) \qquad (3)$$

- Equation 3 can also be rewritten as:

$$f = F_{\mathrm{min}} - \left(\frac{F_{\mathrm{max}} - F_{\mathrm{min}}}{V_{\mathrm{max}} - V_{\mathrm{min}}}\right) \cdot (v_{\mathrm{in}}(t) - V_{\mathrm{min}}) \qquad (4)$$



Voltage Controlled Oscillator (Implementation)

```
module vco (out, in);
    parameter real Vmin=0;                           // minimum input voltage (V)
    parameter real Vmax=Vmin+1 from (Vmin:inf);      // maximum input voltage (V)
    parameter real Fmin=1 from (0:inf);              // minimum output freq (Hz)
    parameter real Fmax=2*Fmin from (Fmin:inf);      // maximum output freq (Hz)
    parameter real ampl=1;                           // output amplitude (V)

    input in; output out;
    voltage out, in;

    real freq, phase;

    analog begin
        // compute the freq from the input voltage
        freq = (V(in) - Vmin)*(Fmax - Fmin) / (Vmax - Vmin) + Fmin;

        // bound the frequency (this is optional)
        if (freq > Fmax) freq = Fmax;
        if (freq < Fmin) freq = Fmin;

        // phase is the integral of the freq modulo 2π
        phase = 2*`M_PI*idtmod(freq, 0.0, 1.0, -0.5);
        V(out) <+ ampl * sin(phase);

        // bound the timestep to at least 10 points per sample
        $bound_step(0.1 / freq);
    end
endmodule
```

Verilog-D

Verilog-D Module Template

```
module module_name (port1, port2, port3, port4);          // 1. Module Definition

    input port1, port2;                                    // 2. Port Definitions
    output [7:0] port3;
    inout port4;

    reg [7:0] R1, M1[1:1024];                              // 3. Datatype Declarations
    wire w1, w2, w3, w4;
    parameter C1 = "this is a string";
    parameter [2:0] S1 = 3'b001,
                    S2 = 3'b010,
                    S3 = 3'b100;


                                                           // 4.  Module Body
    comp U1 (w3, w4);                                      // 4a. Module Instances
    comp U2 (.p1(w3), .p2(w4));

    assign w1 = expression;                                // 4b. Continuous Assignments
    wire (strong1, weak0) [3:0] #(2,3) w2 = expression;

    initial begin : block_name                             // 4c. Procedural Blocks
      // statements
    end

    always begin
      // statements
    end
endmodule
```

Verilog-D Statements

```
wait (expression)
@(a or b or c)
@(posedge clk)

reg = expression; // blocking
#delay reg = expression;
reg = #delay expression;
vector_reg[Bit] = expression;
vector_reg[MSB:LSB] = expression;
memory[address] = expression;
reg ≤ expression; // non-blocking
assign net = expression;
assign #delay net = expression;

begin
    ...
end

fork
    ...
join
```

```
if (condition)
    ...
else if (condition)
    ...
else
    ...

case (selection)
    choice1:
        ...
    choice2, choice3:
        ...
    default:
        ...
endcase

for (i=0; i<MAX; i=i+1)
    ...

repeat (8)
    ...

while (condition)
    ...

forever
    ...
```

Logic Gates and Switches

Multiple Input Gates	**Multiple Output Gates**	**Tristate Gates**	**Pull Gates**	**MOS Switches**	**Bidirectional Switches**
and	buf	bufi0	pullup	nmos	tran
nand	not	bufi1	pulldown	pmos	tranif0
nor	notif0	notif0	pullup	cmos	tranif1
or	notif1	notif1	pulldown	rnmos	rtran
xor				rpmos	rtranif0
xnor				rcmos	rtranif1

- The number of pins for a primitive gate is defined by the number of nets connected to it:
  - The multiple input gates have one output and a variable number of inputs.
  - The multiple output gates have one input and variable number of outputs.
- By definition outputs are listed first, then inputs.

```
and a1 (out, in1, in2);         // 2-input and
nand n2 (out, in1, in2, in3);   // 3-input nand
not (out1, out2, out3, in1);    // 3-output not
```

Integer Literals

Integer literals have the following form:

```
integer_literal ::= <sign>? <size>? <'base>? number

sign            ::= < + | - >                         ==> (default is +)
size            ::= non_zero_digit <digit>*           ==> (default = 32)
base            ::= <b | B | o | O | d | D | h | H>   ==> (default = d)
number          ::= value <value>*
value           ::= <digit | x | X | z | Z | ?>
digit           ::= <0 | non_zero_digit>
non_zero_digit  ::= <1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9>
```

Examples

- `4'b11` is equal to 0011.
- `-5'b00001` is equal to 11111 (2's complement of 00001).

Logic Values and Strength Values

Verilog provides 4 logic values.

Logic Value	**Description**
0	Zero, low or false
1	One, high or true
z or Z	High impedance, tri-stated or floating
x or X	Unknown or uninitialized (logic level is 0 or 1)

Logic Values and Strength Values

Verilog provides 4 logic values.

Logic Value	Description
0	Zero, low or false
1	One, high or true
z or Z	High impedance, tri-stated or floating
x or X	Unknown or uninitialized (logic level is 0 or 1)

Logic `0` and `1` are associated with 8 strength levels. When different drivers with varying strengths drive a single wire, Verilog uses these strength levels to resolve the final value of the net.

Strength Level	Strength Name	Specification Keyword
7	Supply Drive	supply0, supply1
6	Strong Pull	strong0, strong1
5	Pull Drive	pull0, pull1
4	Large Capacitance	large0, large1
3	Weak Drive	weak0, weak1
2	Medium Capacitance	medium0, medium1
1	Small Capacitance	small0, small1
0	Hi Impedance	highz0, highz1

Data Types

Nets

- Net data types (e.g., `wire`, `tri`, `supply0`, `supply1`) represent the **physical connections** between the elements of a design.
- Nets reflect the **value** and **strength** level of the drivers of the net or the capacitance of the net, and do not have a value of their own.
- Nets have a resolution function, which resolves a final value when there are multiple drivers on the net.
- Use a net type for a signal that is driven by a module output, a primitive output, or a continuous assignment.

Data Types

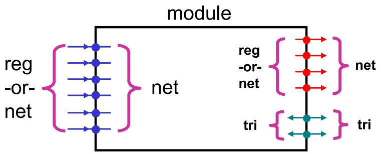
Nets

- Net data types (e.g., `wire`, `tri`, `supply0`, `supply1`) represent the **physical connections** between the elements of a design.
- Nets reflect the **value** and **strength** level of the drivers of the net or the capacitance of the net, and do not have a value of their own.
- Nets have a resolution function, which resolves a final value when there are multiple drivers on the net.
- Use a net type for a signal that is driven by a module output, a primitive output, or a continuous assignment.

Variables

- Variables (e.g., `reg`, `integer`, `real`, `time`) can only be assigned a value from within an initial procedure, an always procedure, a task or a function.
- Variables can only store logic values; they cannot store logic strength.
- Variables are un-initialized at the start of simulation, and will contain a logic `X` until a value is assigned.
- Use a variable type when a signal is assigned a value in a procedure.

Port Connection Rules



- **Inputs:**
  - Internally, inputs must be of **net** data type. The data type term can be omitted if it is **wire**.
  - Externally, inputs may be driven by **net** or **reg** data types.
- **Outputs:**
  - Internally, outputs can be declared as **net** or **reg** data types.
  - Externally, outputs must be connected to a **net** data type.
- **Inouts:**
  - Internally must be of **net** data type, such as **tri**.
  - Externally must be connected to a **net** data type, such as **tri**

Net Type Declaration

The general syntax for declaring a net type is given below:

```
net_type <strength>? <[range]>? <#delay>? net <, net>?;
net_type <strength>? <[range]>? <#delay>? net < = expression>?;

net_type ::= <supply0 | supply1 | tri | triand | trior | wire | wand | wor>
delay    ::= <delay_time | (min:typical:max) | (rise_delay, fall_delay)>
net      ::= identifier <array>?
array    ::= <[integer:integer]>*
```

Examples

```
wire a, b, c;                 // 3 scalar (1-bit) nets
wire [7:0] data_bus;          // 8-bit net
wire [7:0] Q [0:15][0:256];  // 2-dimensional array of 8-bit wires
wire #(2.4, 1.8) carry;       // net with rise, fall delays
```

Variable Type Declaration

The general syntax for declaring a variable type is given below:

```
variable_type <[range]>? var <, var>?;
variable_type <[range]>? var < = expression>?;

variable_type ::= <reg | integer | time | real | realtime>
var           ::= identifier <array>?
array         ::= <[integer:integer]>*
```

Examples

```
reg a, b, c;                   // three scalar (1-bit) variables
reg [7:0] d1, d2;              // two 8-bit variables
reg [7:0] M[0:15][0:256];      // 2-dimensional array of 8-bit variables
reg [7:0] v1, M[0:15][0:256];  // 8-bit variable and a 2-dimensional array of 8-bit variables
```

Assigning Delays

Delays are scaled by the units given in the `` `timescale 1ns/100ps `` directive.

- The first number sets the duration of a unit of time (1ns).
- The second number sets the minimum digital timestep (100ps).

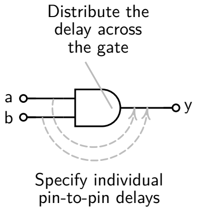
Assigning Delays

Delays are scaled by the units given in the `` `timescale 1ns/100ps `` directive.

- The first number sets the duration of a unit of time (1ns).
- The second number sets the minimum digital timestep (100ps).

Distribute the delay

```
`timescale 1ns/100ps
module and2 (y, a, b);
    input a, b;
    output y;
    and #2.1 (y, a, b);
endmodule
```


Distribute the delay across the gate
a
b
y
Specify individual pin-to-pin delays

Specify pin-to-pin delays

```
`timescale 1ns/100ps
module and2 (y, a, b);
    input a, b;
    output y;
    and (y, a, b);
    specify
        (a=>y) = 2.1;
        (b=>y) = 2.5;
    endspecify
endmodule
```

Procedural Blocks

Procedural blocks can be used to model both combinational and sequential logic.

```
<initial | always> statement

statement       ::= <<timing_control | procedural_statement | statement_group>>*
statement_group ::= <begin | fork> <:group_name>?
                        <local_variable_declarations>?
                        procedural_statements
                    <end | join>
```

Procedural Blocks

Procedural blocks can be used to model both combinational and sequential logic.

```
<initial | always> statement

statement       ::= <<timing_control | procedural_statement | statement_group>>*
statement_group ::= <begin | fork> <:group_name>?
                        <local_variable_declarations>?
                        procedural_statements
                    <end | join>
```

There are two types of procedural blocks:

- **initial blocks**: contain statements that execute only once during initialization.
- **always blocks**: contain statements that execute in a loop.

Procedural Blocks

Procedural blocks can be used to model both combinational and sequential logic.

```
<initial | always> statement

statement       ::= <<timing_control | procedural_statement | statement_group>>*
statement_group ::= <begin | fork> <:group_name>?
                        <local_variable_declarations>?
                        procedural_statements
                    <end | join>
```

There are two types of procedural blocks:

- **initial blocks**: contain statements that execute only once during initialization.
- **always blocks**: contain statements that execute in a loop.

There are two types of statement groups:

- In a **begin–end** group, the statements are evaluated in the order they are listed. Each time control in the group is relative to previous time controls.
- In a **fork–join** group, the statements are evaluated concurrently. Each time control in the group is absolute to the time the group started.

Timing Control

There are three types of **timing control** constructs:

- **Simple delays**: These introduce a fixed delay before executing subsequent statements.
  - Syntax: `#delay variable_identifier = expression;`
- **Wait statements**: Used to halt the execution until a specified condition becomes true.
  - Syntax: `wait(expression)`
- **Event-driven controls**: Uses a sensitivity lists to specify conditions under which statements are executed.
  - Syntax: `@(sensitivity_list)`
  - The **sensitivity list** is used to infer either combinational logic or sequential logic.
    - `always @(signal, signal, ... )` infers combinational logic if the list of signals contains all signals read within the procedure.
    - `always @(posedge signal, negedge signal, ... )` infers sequential logic. Either the positive or negative edge can be specified for each signal in the list.

Procedural Statements

Procedural statements are used to control the execution flow.

- **Conditional Statements**:
  - `if` statment: Executes one block of statements or another based on a condition.
  - `case` statement: Selects one of many blocks of statements to execute, based on the value of an expression.
- **Loop Constructs**:
  - `for`: Executes a block of statements multiple times, with a loop index.
  - `while`: Continues to execute as long as the specified condition remains true.
  - `repeat`: Repeats the enclosed statements a specific number of times.
  - `forever`: Repeats the enclosed statements indefinitely.

Procedural Statements

Procedural statements are used to control the execution flow.

- **Conditional Statements**:
  - `if` statment: Executes one block of statements or another based on a condition.
  - `case` statement: Selects one of many blocks of statements to execute, based on the value of an expression.
- **Loop Constructs**:
  - `for`: Executes a block of statements multiple times, with a loop index.
  - `while`: Continues to execute as long as the specified condition remains true.
  - `repeat`: Repeats the enclosed statements a specific number of times.
  - `forever`: Repeats the enclosed statements indefinitely.

always procedures may have:

- **blocking statement**, with the = operator, which is used to model combinational logic within always blocks.
- **non-blocking statement**, with the $\leq$ operator, which is used to model sequential logic.

Verilog-AMS

Introduction

- **Verilog-AMS** allows the integeration analog and digital behavior within a single model.
  - Models can instantiate a collection of analog, digital, and mixed-signal modules.
  - Models may contain both analog and digital behavioral descriptions.
    - **Digital behavior** is described using initial and always blocks.
    - **Analog behavior** is described within analog blocks.
    - Multiple initial and always blocks can be used, but only one analog block is allowed.

Introduction

- **Verilog-AMS** allows the integeration analog and digital behavior within a single model.
  - Models can instantiate a collection of analog, digital, and mixed-signal modules.
  - Models may contain both analog and digital behavioral descriptions.
    - **Digital behavior** is described using initial and always blocks.
    - **Analog behavior** is described within analog blocks.
    - Multiple initial and always blocks can be used, but only one analog block is allowed.
- Important Verilog-AMS concepts:
  - **Domain**: Depending on the computational methods used to calculate the values of a signals or variable, the signal or variable can belong to the analog domain (also known as continuous domain) or digital domain (also known as discrete domain).
  - **Context**: Statements that appear in an analog block are in the continuous context; statements in any other location are in the discrete context. A particular variable can be assigned values in either context, but not in both contexts.

Introduction

- **Verilog-AMS** allows the integeration analog and digital behavior within a single model.
    - Models can instantiate a collection of analog, digital, and mixed-signal modules.
    - Models may contain both analog and digital behavioral descriptions.
        - **Digital behavior** is described using initial and always blocks.
        - **Analog behavior** is described within analog blocks.
        - Multiple initial and always blocks can be used, but only one analog block is allowed.
- Important Verilog-AMS concepts:
    - **Domain**: Depending on the computational methods used to calculate the values of a signals or variable, the signal or variable can belong to the analog domain (also known as continuous domain) or digital domain (also known as discrete domain).
    - **Context**: Statements that appear in an analog block are in the continuous context; statements in any other location are in the discrete context. A particular variable can be assigned values in either context, but not in both contexts.
- The signals and variables of each domain can be referenced in the other context.
    - **Read** operations of nets and variables in both domains are allowed from both contexts.
    - **Write** operations of nets and variables are only allowed from within the context of their domain.

Behavioral Interaction Example

```
`timescale 1ns/1ns
module mod (in);
    integer abve;           // Will be an analog-owned variable.
    integer below;          // Will be an analog-owned variable.
    integer d;              // Will be a digital-owned variable.

    electrical in;

    always begin            // Enter the digital context.
        if ( abve )         // Read the analog variable in the digital context.
            d = 1;          // Write the variable d in the digital context.
        if ( below )
        d = 0;              // d, because written in digital context, is owned by digital.
        #5;
    end

    analog begin            // Enter the analog context.
        @(cross (V(in) - 2.5, +1))
            abve = 1;       // Write to the variable abve in the analog context.
        @(cross (V(in) - 2.5, -1))
            abve = 0;       // abve, because written in analog context,is owned by analog.
        if (d == 1)         // Read the value of d in the analog context.
            $strobe("d is still high\n");
    end
endmodule
```

Discipline Declaration

- Verilog-AMS uses the concept of **disciplines** to differentiate between analog and digital signals.
  - Disciplines are not part of Verilog-D.
  - The `` `default_discipline `` directive allows pre-existing Verilog-D modules to be used without needing to add discipline declarations for all digital signals.
  - The `` `default_discipline `` directive could be set to a user defined discipline. The default is `logic`.

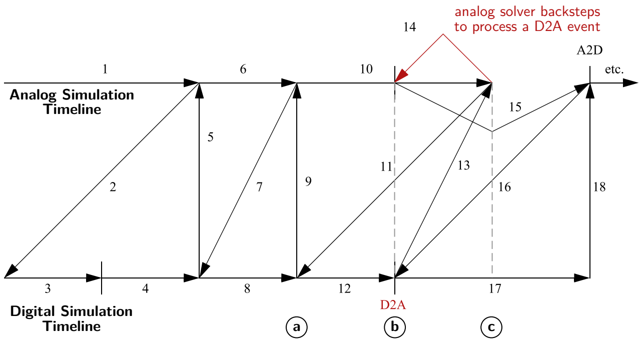
```
`include "disciplines.vams"
`default_discipline logic  // this is already the default
module onebit_dac (in, out);
    input in; inout out;
    electrical out; // discipline declaration for analog signals is required
    logic in;       // discipline declaration for digital signals is optional
    wire in;        // default digital net type is wire, so this line is also optional
    real vout;
    analog
        if (in==0)
            vout = 0.0;
        else
            vout 3.0;
    V(out) <+ vout;
endmodule
```

The Mixed-Signal Synchronization Cycle

Verilog-AMS enables synchronization between the discrete and continuous domains.

- The analog solver can backstep, allowing it to go back in simulation time.
- Solver engines synchronize using a master/slave relationship:
  - The analog solver acts as the master, leading in simulation time.
  - The digital solver is the slave, lagging in simulation time.
- At $t = 0$, control is exchanged between digital and analog solvers until a stable DC solution is found. The sequence is as follows:
  - The analog solver initializes all signals based on the analog initial block.
  - The digital solver processes all initial blocks and digital states.
  - The analog solver updates the analog solution based on the digital values.
  - The digital solver adjusts states influenced by the previous analog values.
  - Finally, the analog solver settles on values best influenced by last digital states to leave a $t = 0$ timestep with a stable AMS DC solution for transient.

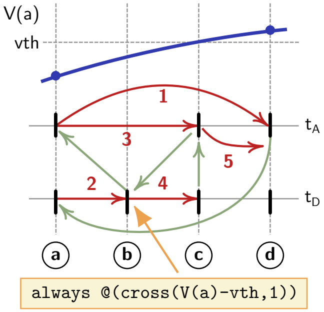
Verilog-AMS Synchronization Algorithm



analog solver backsteps to process a D2A event
14
A2D
1
6
10
etc.
Analog Simulation Timeline
5
15
2
7
9
11
13
16
18
3
4
8
12
17
D2A
Digital Simulation Timeline
a
b
c

- The analog solver always leads in simulation time.
- At time (a), the analog solver advances to time (c) and then transfers control to the digital solver, which remains at time (a).
- The digital solver then advances to time (b), where a Digital-to-Analog (D2A) event occurs, meaning a digital value is used in an analog process.
- The analog solver backsteps to time (b) to incorporate the digital state at that time.

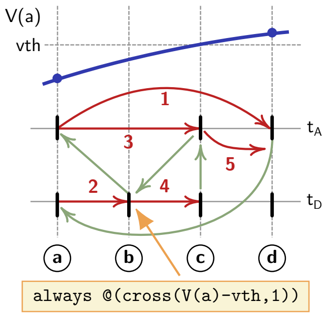
Analog Backstepping

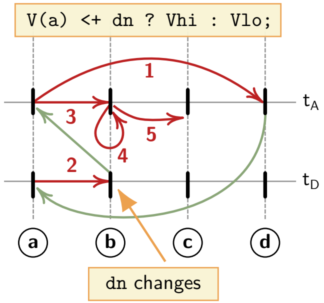


```
always @(cross(V(a)-vth,1))
```

The `@(cross())` function in the digital context forces the analog solver to backstep from time ⓓ to time ⓐ, and then proceed to time ⓒ, where the crossing event occurs.

Analog Backstepping



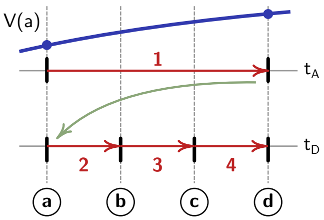


The `@(cross())` function in the digital context forces the analog solver to backstep from time ⓓ to time ⓐ, and then proceed to time ⓒ, where the crossing event occurs.

- When an analog signal is sensitive to a digital signal, and the digital signal changes, the analog solver has to backstep to incorporate the changes.
- If the digital signal can cause the analog signal to change instantaneously, then the use of the transition function is strongly recommended to smooth out the changes.

Accessing Analog Signals Inside A Digital Context

- If an analog process is sensitive to a digital signal, then it is evaluated at the point when the digital signal changes.
- If a digital process is sensitive to an analog signal, then since the analog solver is ahead of the digital solver, Verilog-AMS does not require the analog solver to place a timepoint at the instant when the analog value is sensed. As a result, the analog signal is interpolated at intermediate timesteps.



Connect Modules

- When two modules are connected, the conversion of signals between the analog and digital domains is handled by interface blocks called **connect modules**, which can either be inserted manually, or automatically by the simulator, at the interface between analog and digital nets.

```
connectmodule connect_module_name (port1, port2);
    module_body
endmodule
```

Connect Modules

- When two modules are connected, the conversion of signals between the analog and digital domains is handled by interface blocks called **connect modules**, which can either be inserted manually, or automatically by the simulator, at the interface between analog and digital nets.

```
connectmodule connect_module_name (port1, port2);
    module_body
endmodule
```

- The selection of the appropriate connect module for automatic insertion at an interface interface (d2a, a2d, bidirectional) is determined by the **connect rule**, which relies on the disciplines and port directions of the connected nets.

```
connectrules connect_rule_name;
    <connect_statement>*
endconnectrules

connect_statement ::= connect connect_module_name <<merged | split>>?
<#(param_list)>? <<dir>?< discipline_name <, <dir>? discipline_name>*>?>?;
```

Connect Modules

- When two modules are connected, the conversion of signals between the analog and digital domains is handled by interface blocks called **connect modules**, which can either be inserted manually, or automatically by the simulator, at the interface between analog and digital nets.

```
connectmodule connect_module_name (port1, port2);
    module_body
endmodule
```

- The selection of the appropriate connect module for automatic insertion at an interface interface (d2a, a2d, bidirectional) is determined by the **connect rule**, which relies on the disciplines and port directions of the connected nets.

```
connectrules connect_rule_name;
    <connect_statement>*
endconnectrules

connect_statement ::= connect connect_module_name <<merged | split>>?
<#(param_list)>? <<dir>?< discipline_name <, <dir>? discipline_name>*>?>?;
```

- If a net does not have an explicit discipline definition, the **discipline resolution** algorithm is called to resolve and assign a discipline based on the discipline of other nets connected to it.

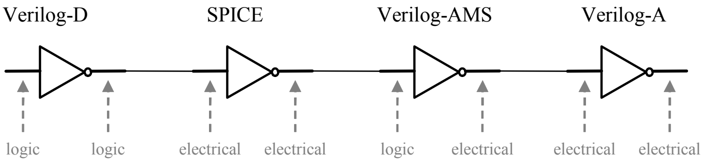
Connect Module Modeling

- Mixed signal simulations must mimic real circuit behavior closely.
  - Analog considerations: impedance, nonlinearity, supply voltage dependency, etc.
  - Digital considerations: rise/fall times, digital supply voltage dependencies, etc.

```
connectmodule a2d (dout, ain);
    input ain; electrical ain;
    output dout; logic dout; reg dout;
    parameter real VH = 2.7;
    parameter real VL = 0.5;
    parameter real C = 100f;

    // load capacitance
    analog
        I(ain) <+ C*ddt(V(ain));

    always @(above(V(ain) - VH)
        dout = 1'b1;

    always @(above(VL - V(ain))
        dout = 0'b0;

endmodule
```

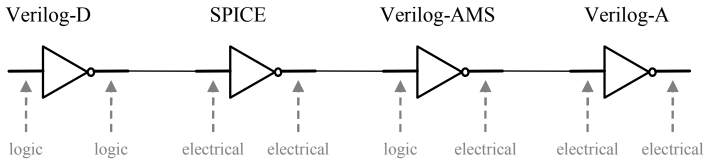
```
connectmodule d2a (aout, din);
    input din; logic din;
    output aout; electrical aout;
    parameter real VH=5.0 from [0:inf);
    parameter real VL=0.0 from [0:inf);
    parameter real ROUT=1k from (0:inf);
    parameter real TR=1n from [0:inf);
    parameter real TF=1n from [0:inf);
    real val;

    analog begin
        // driver voltage
        val = (din == 1b'1) ? VH : VL;
        V(aout) <+ transition(val, TR, TF);

        // driver resistance
        V(aout) <+ I(out) * ROUT;
    end
endmodule
```

Connect Modules Insertion

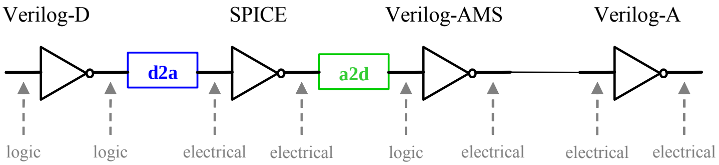


Before automatic connect modules insertion

Connect Modules Insertion



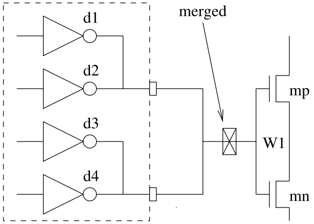
Before automatic connect modules insertion

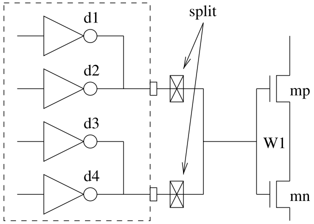


After automatic connect modules insertion

Connect Module Mode

Connect rules have a **connect mode** attribute for scenarios where a net from one discipline connects to several ports of another discipline.





Merged Mode:

- All connections are combined into a single connect module.
- This connect module then connects to each individual digital port.
- This is the default connect mode.

Split Mode:

- An individual connect module is inserted at each digital port.
- Provides higher accuracy since the load of each digital port on the analog side is accurately represented.
- Results in slower simulation speed due to larger number of connect modules.

Discipline Resolution

- Verilog-AMS allows for **undeclared** net disciplines in connectivity statements.
- The **discipline resolution algorithm** determines the appropriate disciplines for undeclared nets.
- Verilog-AMS provides two discipline resolution algorithms: basic (default) and detailed.

Discipline Resolution

- Verilog-AMS allows for **undeclared** net disciplines in connectivity statements.
- The **discipline resolution algorithm** determines the appropriate disciplines for undeclared nets.
- Verilog-AMS provides two discipline resolution algorithms: basic (default) and detailed.

Basic Method

- Propagates net disciplines up the hierarchy.
- Net takes the discipline of lower level ports if all connected ports share the same discipline.
- Requires user-defined resolution rule if lower level ports share the same domain but have different disciplines.
- If one lower level port is analog, the higher level net becomes analog and a connect module is needed.

Discipline Resolution

- Verilog-AMS allows for **undeclared** net disciplines in connectivity statements.
- The **discipline resolution algorithm** determines the appropriate disciplines for undeclared nets.
- Verilog-AMS provides two discipline resolution algorithms: basic (default) and detailed.

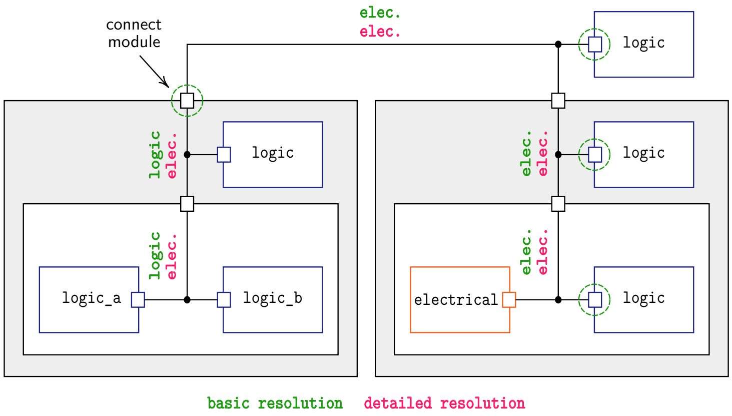
Basic Method

- Propagates net disciplines up the hierarchy.
- Net takes the discipline of lower level ports if all connected ports share the same discipline.
- Requires user-defined resolution rule if lower level ports share the same domain but have different disciplines.
- If one lower level port is analog, the higher level net becomes analog and a connect module is needed.

Detailed Method

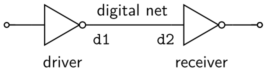
- Bottom-up traversal propagates analog disciplines upwards.
- Top-down traversal propagates analog disciplines downwards.
- Every net that in any way connects to an analog port becomes analog.

Discipline Resolution Example



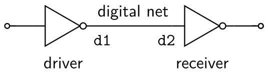
Driver-Receiver Segregation

- In a purely digital net, drivers generate signals that propagate directly to receivers.



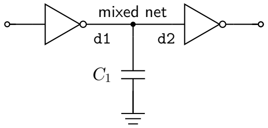
Driver-Receiver Segregation

- In a purely digital net, drivers generate signals that propagate directly to receivers.



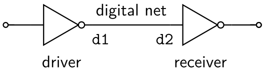
- Adding an analog capacitor to the circuit can affect the propagation of the digital signals.

- This also turns the net between `d1` and `d2` into a **mixed net**.

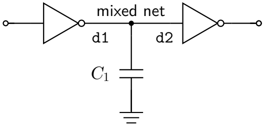


Driver-Receiver Segregation

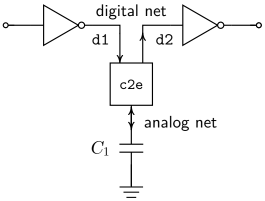
- In a purely digital net, drivers generate signals that propagate directly to receivers.



- Adding an analog capacitor to the circuit can affect the propagation of the digital signals.
- This also turns the net between d1 and d2 into a **mixed net**.



- Because the net is mixed, it is subject to **driver-receiver segregation**, which severs the direct connection between d1 and d2 and inserts a c2e connect module.



```
module c2e(d, a);
    input d; logic d;
    output a; electrical a;
    assign d = d; // Both read and drive the
    ↪  digital port
    analog // Perform digital to analog translation
        V(a) <+ transition( d == 1 ? 5.0 : 0.0 );
endmodule
```

Verilog-AMS Simulation

AMS Simulation With Cadence Tools

- Simulating mixed-signal designs from the command-line is done using the `xrun` command.
- `xrun` takes files from different simulation languages and compiles them using the appropriate compilers. It then invokes `xmelab` to elaborate the design and generate a simulation snapshot. Finally, it invokes the `xmsim` simulator to simulate the snapshot.
- Simulation files used for `xrun` include the following:
    - Source files (e.g. verilog-ams, verilog, vhdl, spice, etc.,).
    - AMS control file (`.scs`), where the simulation is configured.
    - Probe definition file (`.tcl`), where the signals to be saved are defined.

```
xrun <source_files>+ control.scs -input probe.tcl <←amsconnrules name>+>?
←timescale 1ns/1ns>? ←iereport>? ←access +rwc>? ←gui>?
```

- The `timescale` option sets the default timescale.
- `-iereport` tells `xrun` to report the connect modules inserted during the elaboration phase.
- `-access` provides read access to the simulation database for SimVision.

Voltage Controlled Oscillator

The code below shows the Verilog-AMS description of a VCO in which the analog input voltage controls the frequency of an output clock.

- Since the time scale is 1ns, the delay is multiplied by 1e9.
- The analog input voltage is sampled at half the period of the output clock.

vco.vams

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

module vco (out, in);
    parameter real f0 = 100k from (0:inf);  // center frequency (Hz)
    parameter real kvco = 10k;              // gain (Hz/V)
    input in; electrical in;
    output out; logic out; reg out;
    real vin;

    initial out = 0;

    always begin
        vin = V(in);
        #(0.5e9 / (f0 + kvco * vin))
        out = ~out;
    end
endmodule
```

Simulation

```
xrun testbench.vams vco.vams control.scs -input probe.tcl -access +rwc -gui
```

testbench.vams

```
`timescale 10ns / 10ps
`include "disciplines.vams"

module testbench ();
    electrical gnd;
    ground gnd;
    wire out;

    vco #(.f0(100M), .kvco(10M)) vco0 (out, in, gnd);
    vsource #(.type("sine"), .ampl(0.5), .dc(0.5), .freq(1M)) v0 (in, gnd);
endmodule
```

probe.tcl

```
database -open waves -into waves.shm -default
probe -create testbench -depth all -shm -waveform
run 100us
exit
```

control.scs

```
*
transient tran stop=100us
```