```
module shiftPlus5(in, out);
input in;
output out;
voltage in, out;  //voltage is a signal flow
                  //discipline compatible with
                  //electrical, but having a
                  //potential nature only
analog begin
   V(out) <+ 5.0 + V(in);
end
endmodule
```

When a net of signal flow discipline with potential nature only is bound to a conservative node(ex: electrical), contributions made to that net behave as **voltage sources** to ground.

The following example is a current mirror:

```
module currmir(in, out);
   input in;
   output out;
   current in, out;  // current is a signal flow
                     // discipline compatible with
                     // electrical, but having a
                     // flow nature only

   analog begin
      I(out) <+ -I(in);
   end
endmodule
```

When a net of signal flow discipline with flow nature only is bound to a conservative node, contributions made to that net behave as **current sources**.

Inout : not allowed for sign-flow discipline (only in or out: see above code)

Users typically use signal-flow models early in the design cycle when the system is described in abstract terms, and gradually convert component models to conservative form as the design progresses.

Signal-flow ports only require the type of either potential or flow to be specified, whereas

```
    analog begin
       I(out) <+ -I(in);
    end
endmodule
```

When a net of signal flow discipline with flow nature only is bound to a conservative node, contributions made to that net behave as **current sources**.

Inout : not allowed for sign-flow discipline (only in or out: see above code)

Users typically use signal-flow models early in the design cycle when the system is described in abstract terms, and gradually convert component models to conservative form as the design progresses.

Signal-flow ports only require the type of either potential or flow to be specified, whereas conservative ports require types for both values (the potential and flow).

For example, consider a differential voltage amplifier, a differential current amplifier, and a resistor. The amplifiers are written using signal-flow ports and the **resistor uses conservative ports.**

```
module voltage_amplifier (out, in);
   input in;
   output out;
   voltage  out,      // Discipline voltage defined elsewhere
            in;       // with access function V()
   parameter real GAIN_V = 10.0;


  analog
     V(out) <+ GAIN_V * V(in);

endmodule
```

In this case, only the voltage on the ports are declared because only voltage is used in the body of the model

```
module current_amplifier (out, in);
   input in;
   output out;
   current  out,      // Discipline current defined elsewhere
            in;       // with access function I()
   parameter real GAIN_I = 10.0;

   analog
      I(out) <+ GAIN_I * I(in);

endmodule
```

Here, only current is used in the body of the model, so only current need be declared at the ports.

```
module resistor (a, b);
   inout a, b;
   electrical a, b;          // access functions are V() and I()
   parameter real R = 1.0;

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

endmodule
```

The description of the resistor relates both the voltage and current on the ports. Both are defined in the conservative discipline `electrical`.

In summary, only those signals types declared on the ports are accessible in the body of the model. Conversely, only those signals types used in the body need be declared.

Declare voltage, only V() ; declare current , only I() ; declare electrical , both V(), I().

The underscore character (_) shall be legal anywhere in a number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

Example 1 — Unsized Constant numbers

```
659       // is a decimal number
'h 837FF  // is a hexadecimal number
'o7460    // is an octal number
4af       // is illegal (hexadecimal format requires 'h)
```

Example 2 — Sized constant numbers

```
4'b1001   // is a 4-bit binary number
'D 3      // is a 5-bit decimal number
3'b01x    // is a 3-bit number with the least
          // significant bit unknown
12'hx     // is a 12-bit unknown number
16'hz     // is a 16-bit high-impedance number
```

Example 3 — Using sign with constant numbers

```
8 'd -6   // this is illegal syntax
-8 'd 6   // this defines the two's complement of 6,
          // held in 8 bits—equivalent to -(8'd 6)
4 'shf    // this denotes the 4-bit number '1111', to
          // be interpreted as a 2's complement number,
          // or '-1'. This is equivalent to -4'h 1
-4 'sd15 // this is equivalent to -(-4'd 1), or '0001'
16'sd?    // the same as 16'sbz
```

Example 4 — Automatic left padding

```
reg [11:0] a, b, c, d;
initial begin
   a = 'h x;       // yields xxx
   b = 'h 3x;      // yields 03x
   c = 'h z3;      // yields zz3
   d = 'h 0z3;     // yields 0z3
end
```

Example 4 — Automatic left padding

```
reg [11:0] a, b, c, d;
initial begin
   a = 'h x;        // yields xxx
   b = 'h 3x;       // yields 03x
   c = 'h z3;       // yields zz3
   d = 'h 0z3;      // yields 0z3
end
reg [84:0] e, f, g;
   e = 'h5;         // yields {82{1'b0},3'b101}
   f = 'hx;         // yields {85{1'hx}}
   g = 'hz;         // yields {85{1'hz}}
```

Example 5 — Using underscore character in numbers

```
27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001
```

Sized negative constant numbers and sized signed constant numbers are sign-extended when assigned to a `reg` data type, regardless of whether or not the `reg` itself is signed.

The default length of `x` and `z` is the same as the default length of an integer.

Parameters represent constants, hence it is illegal to modify their value at runtime.

```
parameter real slew_rate = 1e-3;
parameter integer size = 16;
```

3.5 Genvars

Genvars are integer-valued variables which compose static expressions for instantiating structure behaviorally such as accessing analog signals within behavioral looping constructs.

Examples:

```
genvar i;
analog begin
   ...
   for (i = 0; i < 8; i = i + 1) begin
      V(out[i]) <+ transition(value[i], td, tr);
   end
   ...
end
```

3.6 Net_discipline

Net_discipline is used to declare analog nets, as well as declaring the domains of digital nets and regs.

A discipline is characterized by the domain and the attributes defined in the natures for potential and flow.

3.6.1 Natures

A nature is a collection of attributes. In Verilog-AMS HDL, there are several pre-defined attributes. In addition, user-defined attributes can be declared and assigned constant values in a nature.

A nature shall be defined between the keywords **nature** and **endnature**. Each nature definition shall have a unique identifier as the name of the nature and shall include all the required attributes specified in 3.6.1.2.

Examples:

```
nature current;
   units = "A";
   access = I;
   idt_nature = charge;
   abstol = 1u;
```

3.6.1 Natures

A nature is a collection of attributes. In Verilog-AMS HDL, there are several pre-defined attributes. In addition, user-defined attributes can be declared and assigned constant values in a nature.

A nature shall be defined between the keywords **nature** and **endnature**. Each nature definition shall have a unique identifier as the name of the nature and shall include all the required attributes specified in 3.6.1.2.

Examples:

```
nature current;
   units = "A";
   access = I;
   idt_nature = charge;
   abstol = 1u;
endnature

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

3.6.2 Disciplines

A *discipline* description consists of specifying a **domain** type and binding any *natures* to **potential** or **flow**.

Examples:

Conservative discipline

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

Signal-flow disciplines

```
discipline voltage;
   potential Voltage;
enddiscipline

discipline current;
   flow Current;
enddiscipline
```

3.7 Real net declarations

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

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

Examples:

```
module drv(in, out);
   input in;
```

Examples:

```
module drv(in, out);
   input in;
   output out;
   wreal in;
   electrical out;
   analog begin
      V(out) <+ in;
   end
endmodule

module top();
   real stim;
   electrical load;
   wreal wrstim;
   assign wrstim = stim;
   drv f1(wrstim, load);
   always begin
      #1 stim = stim + 0.1;
   end
endmodule
```

3.12.1 Port Branches

A port branch is a special type of branch used to access the flow into a port of a module (see 5.4.3). It is a branch between the upper and lower connections of the port. A port branch is a scalar branch if the port identifier is a scalar port. A port branch is a vector branch if the port identifier is a vector port.

Example:

```
module current_sink(p);
   electrical p;
   branch (<p>) probe_p;
   analog
      $strobe("current probed is %g", I(probe_p));
endmodule
```

4.2.2 Operator precedence

The precedence order of *operators* is shown in Table 4-3.

Table 4-3—Precedence rules for operators

+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (bitwise)	
^ ^~ ~^ (bitwise)	
| (bitwise)	
&&	
|| (logical) or (event) , (event)	

Table 4-3—Precedence

+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (bitwise)	
^ ^~ ~^ (bitwise)	
| (bitwise)	
&&	
|| (logical) or (event) , (event)	

Table 4-3—Precedence rules for operators *(continued)*

?: (conditional operator)	
{} {{}}	Lowest precedence

What is branch?

A module is allowed to access the potential and flow of a branch in another module instance using an access function providing that value is available in the other instance. If it is not available, then an error shall be reported. Reasons why it would be unavailable are:

— The branch does not exist in the other instance

— The access function is not the valid access function for that named branch

An example of a hierarchical access of the potential of a named branch is:

```
module top;
   A a1();
   B b1();
endmodule

module A;
   electrical n,p;
   branch (n,p) b;
   analog V(b) <+ 1.34;
```

```
module top;
   A a1();
   B b1();
endmodule

module A;
   electrical n,p;
   branch (n,p) b;
   analog V(b) <+ 1.34;
endmodule


   module B;
      analog $strobe("voltage == %g", V(top.a1.b));
   endmodule
```

To access an existing unnamed branch in another module instance, the *hierarchical_unnamed_branch_reference* syntax is used.

Example:

```
   analog begin
      // strobes the voltage of the unnamed branch between
      // nets a and b in top.drv.
      $strobe("Voltage == %g", V(top.drv.branch(a,b)));

      // strobes the current flowing through the unnamed port
      // branch for the port p in top.drv
      $strobe("Current == %g", I(top.drv.branch(<p>)));
   end
```

5.6.1 Direct branch contribution statements

The direct contribution statement uses the *branch contribution operator* <+ to describe the mathematical relationship between one or more analog nets within the module. The mapping is done with contribution statements using the form shown in Syntax 5-5:

```
contribution_statement ::= branch_lvalue <+ analog_expression ;          // from A.6.10
branch_lvalue ::= branch_probe_function_call                             // from A.8.5
branch_probe_function_call ::=                                           // from A.8.2
        nature_access_function ( branch_reference )
      | nature_access_function ( analog_net_reference [ , analog_net_reference ] )
```

Syntax 5-5—Syntax for branch contribution

In general, a branch contribution statement consists of two parts, a left-hand side and a right-hand side, separated by a branch contribution operator. The right-hand side can be *analog_expression* can be any combination of linear, nonlinear, or differential expressions of module signals, constants, and parameters which evaluates to or can be promoted to a real value. The left-hand side specifies the source branch signal where the right-hand side shall be assigned. It shall consist of a signal access function applied to a branch.

```
I(n1, n2) <+ expression;
```

```
V(n1, n2) <+ expression;
```

```
module resistor(p, n);
    inout p, n;
    electrical p, n;
    branch (p,n) path;     // named branch
    parameter real r = 0;
```

```
module resistor(p, n);
    inout p, n;
    electrical p, n;
    branch (p,n) path;      // named branch
    parameter real r = 0;

    analog
        V(path) <+ r*I(path);
endmodule

module capacitor(p, n);
    inout p, n;
    electrical p, n;
    parameter real c = 0;

    analog
        I(p,n) <+ c*ddt(V(p, n)); // unnamed branch p,n
endmodule
```

V and I problem? What if V already declared? V() is not valid!! ; Use potential() , flow()

The **potential** and **flow** access functions can also be used to contribute to the potential or flow of a named or unnamed branch. The example below demonstrates the **potential** access functions being used to contribute to a branch and the **flow** and **potential** access functions being used to probe branches. Note V and I cannot be used as access functions because there are parameters called V and I declared in the module.

```
module measure2(p);
   output p;
   electrical p;
   parameter real V = 1.1;
   parameter real I = 1u;
   parameter real R = 10k;
   analog begin
      potential(p) <+ flow(p) * R; // create a resistor
      $strobe("voltage ratio at port 'p' is %g", potential(p) / V);
      $strobe("current ratio through port 'p' is %g", flow(<p>) / I);
   end
endmodule
```

When solving an analog block during an iteration, multiple contributions to the same potential branch or same flow branch will be additive. However, contributing a flow to a branch which already has a value retained for the potential results in the potential being discarded and the branch being converted to a flow source. Conversely, contributing a potential to a branch which already has a value retained for the flow results in the flow being discarded and the branch being converted into a potential source. Unlike variables, the contributed value for a branch is only valid for the current iteration. If a branch is not contributed to, directly or indirectly, for any particular iteration, and it is not a branch probe, it shall be treated as a flow source with a value of 0.

Example 1:

Example 1:

```
if (closed)
    V(p,n) <+ 0;
```

is equivalent to

```
if (closed)
    V(p,n) <+ 0;
else
    I(p,n) <+ 0;
```

Example 2:

The value retention rules specify that the example below will result in an assignment of 7.0 to the potential source for the unnamed branch between ports p and n.

```
module value_ret(p, n);
   inout p, n;
   electrical p, n;
   analog begin
      V(p,n) <+ 1.0; // no previously-retained value, 1 is retained
      I(p,n) <+ 2.0; // potential discarded; flow of 2 retained
      V(p,n) <+ 3.0; // flow discarded; potential of 3 retained
      V(p,n) <+ 4.0; // 4 added to previously-retained 3
   end
endmodule
```

Example 3:

The following module defines a current-controlled current source. Because the branch flow I(ps,ns) appears in an expression on the right-hand side, 5.4.2.1 states that this unnamed branch is a probe and its potential is zero (0).

```
module cccs (p, n, ps, ns);
   inout p, n, ps, ns;
   electrical p, n, ps, ns;
   parameter real A = 1.0;
   analog begin
      I(p,n) <+ A * I(ps,ns);
   end
endmodule
```

```
endmodule
```

Resistor model :: Current source or voltage source??

Voltage source here...

```
analog begin
    V(res) <+ R * I(res);
end
```

Current source here....

```
analog begin
    I(cond) <+ G * V(cond);
end
```

5.6.4 RLC circuits

A series RLC circuit is formulated by summing the voltage across its three components,

$$v(t) = Ri(t) + L\frac{\mathrm{d}}{\mathrm{d}t}i(t) + \frac{1}{C}\int_{-\infty}^{t} i(\tau)d\tau$$

which can be defined as

```
V(p, n) <+ R*I(p, n) + L*ddt(I(p, n)) + idt(I(p, n))/C;
```

A parallel RLC circuit is formulated by summing the currents through its three components,

$$i(t) = \frac{v(t)}{R} + C\frac{\mathrm{d}}{\mathrm{d}t}v(t) + \frac{1}{L}\int_{-\infty}^{t} v(\tau)d\tau$$

which can be defined as

```
I(p, n) <+ V(p, n)/R + C*ddt(V(p, n)) + idt(V(p, n))/L;
```

Just define V for series, and I for parallel ....

UVM::

create() method– Example

class ethernet_agent extends uvm_agent;

`uvm_component_utils(ethernet_agent)-------→ step1

UVM::

create() method– Example

```
class ethernet_agent extends uvm_agent;
`uvm_component_utils(ethernet_agent)-------→ step1

ethernet_driver d1;
ethernet_sequencer s1;
ethernet_monitor m1;

virtual function build_phase (uvm_phase phase);
    d1=ethernet_driver::type_id::create("d1", this);------→step3
    s1=ethernet_sequencer::type_id::create("s1", this);
    m1=ethernet_monitor::type_id::create("m1", this);
endfunction

endclass
```

UVM Factory

- uvm_factory is used to create UVM objects and components.
- Factory Registration:
  - *`uvm_object_utils(T)* – uvm_objects
  - *`uvm_component_utils(T)* - uvm_components

- Factory Overriding: - Two Types
  - Set_type_override_by_type
  - Set_inst_override_by_type

Create invokes function new() and constructs according to the factory database

- Factory provides a *create()* function
  - *type_name::type_id::create(string name, uvm_component parent)*

- Objects are constructed dynamically
- User can alter the behavior of the pre-build code without modifying the code.

Component Overriding – Examples

Base class

```
class ethernet_driver extends uvm_driver;

endclass
```

Derived class1

```
class packet_driver extends etherent_driver;

endclass
```

Derived class2

```
class token_driver extends etherent_driver;

endclass
```

- Pure SystemVerilog code

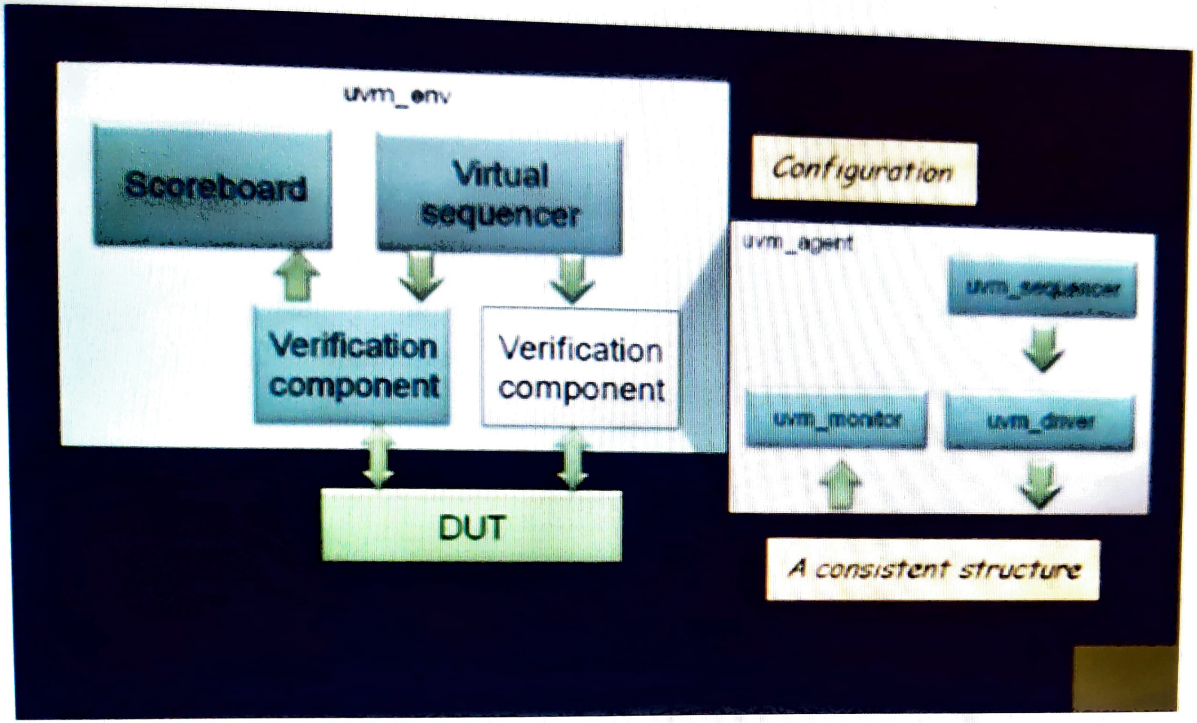
endclass

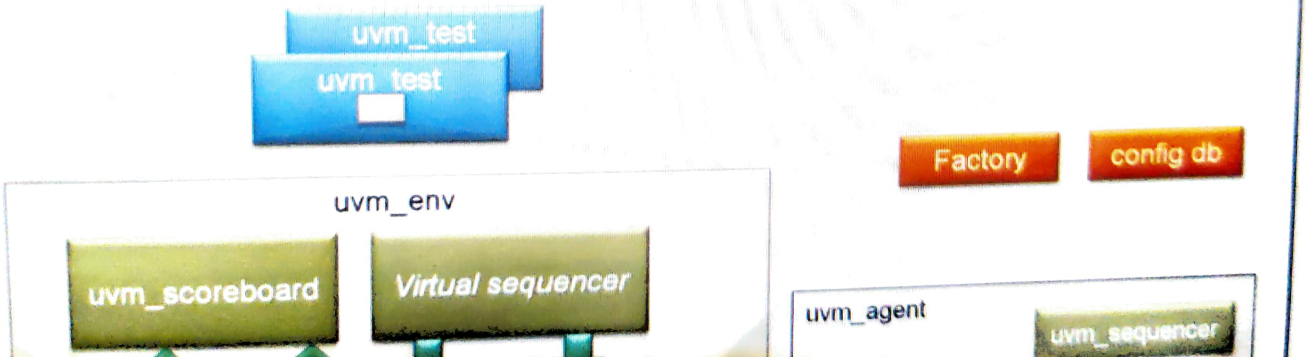
Derived class2

class token_driver extends etherent_driver;

endclass

- Pure SystemVerilog code
- Apache 2.0 open-source license
- Reference Manual and User Guide
- http://www.accellera.org/activities/vip/


uvm_env
Scoreboard
Virtual sequencer
Verification component
Verification component
DUT
Configuration
uvm_agent
uvm_sequencer
uvm_monitor
uvm_driver
A consistent structure


uvm_test
uvm_test
Factory
config db
uvm_env
uvm_scoreboard
Virtual sequencer
uvm_agent
uvm_sequencer