2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 27-MARCH 2, 2023

Take AIM!
Introducing the Analog Information Model

Chuck McClish

Microchip Technology Inc.

MICROCHIP

accellera SYSTEMS INITIATIVE

Motivation for Methodology

- SystemVerilog User Defined Nettypes (UDNs) have drawbacks:
  - Unable to connect dissimilar types
  - Unable to handle current calculations within the resolution function
  - No support for specify blocks
  - Vendor specific AMS cosimulation setup
  - Developing power aware models using UDNs and UPF are cumbersome
- The Analog Information Model (or AIM) provides a solution to these issues, while making modeling easier

AIM Nodes and Connectors

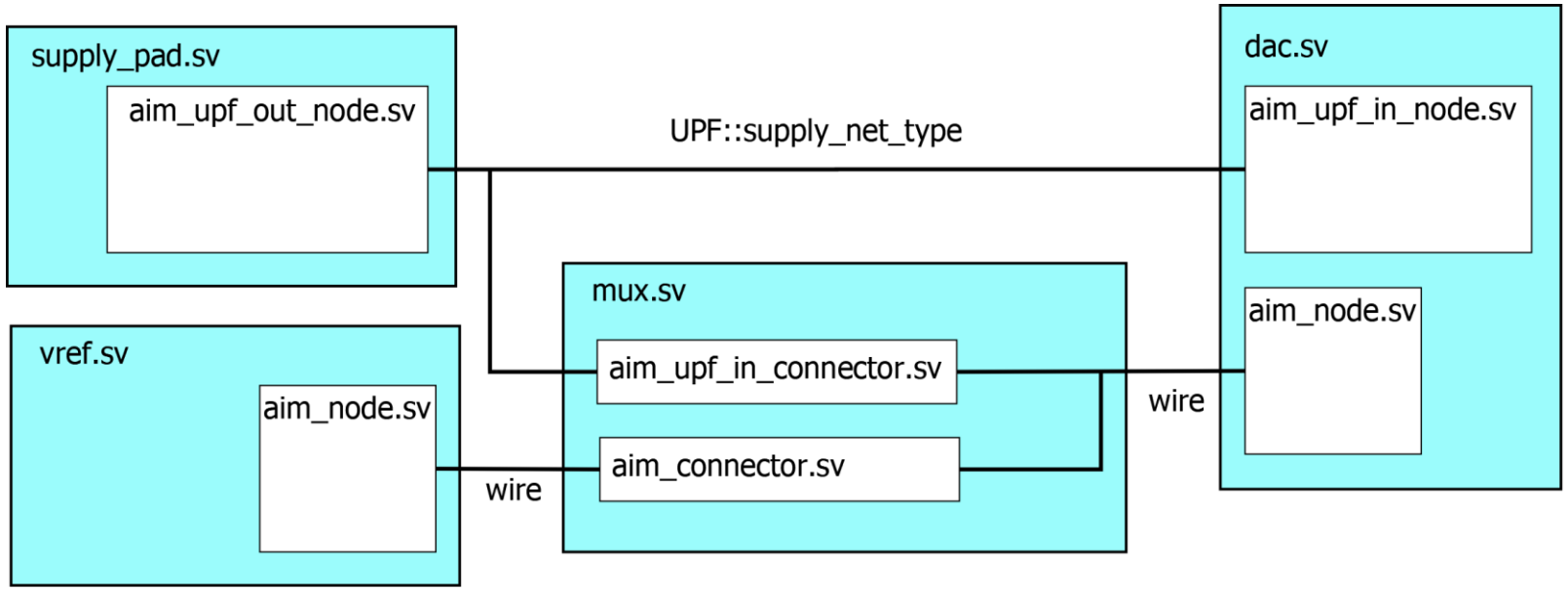
- AIM nodes used as endpoints
  - Properties and methods
- AIM connectors used as `tran` gates to connect 2 nets together
  - Embedded ‘en’ variable can be driven to connect/disconnect
- Different flavors of nodes and connectors based on net type

```
module aim_node import aim_pkg::*; (inout wire net);

  // Properties
  bit    is_ok;
  real   voltage;
  real   current;
  real   resistance;
  node_t node_type; //From aim_pkg

  // Methods (psuedo code prototype declarations)
  task automatic set_voltage(real v);
  task automatic set_current(real v);
  task automatic set_resistance(real v);
  task automatic set_node_type(node_t node_type);
endmodule
```

```
module aim_connector import aim_pkg::*; (inout wire a, inout wire b);
  wire (weak0,weak1) en = 1;
  tranif1 (a, b, en);
endmodule
```

```
module aim_upf_in_node import UPF::*; (input UPF::supply_net_type net);
endmodule

module aim_upf_out_node import UPF::*; (output UPF::supply_net_type net);
endmodule

module aim_upf_in_connector import UPF::*; (input UPF::supply_net_type a, output wire b);
endmodule

module aim_upf_out_connector import UPF::*; (input wire a, output UPF::supply_net_type b);
endmodule
```

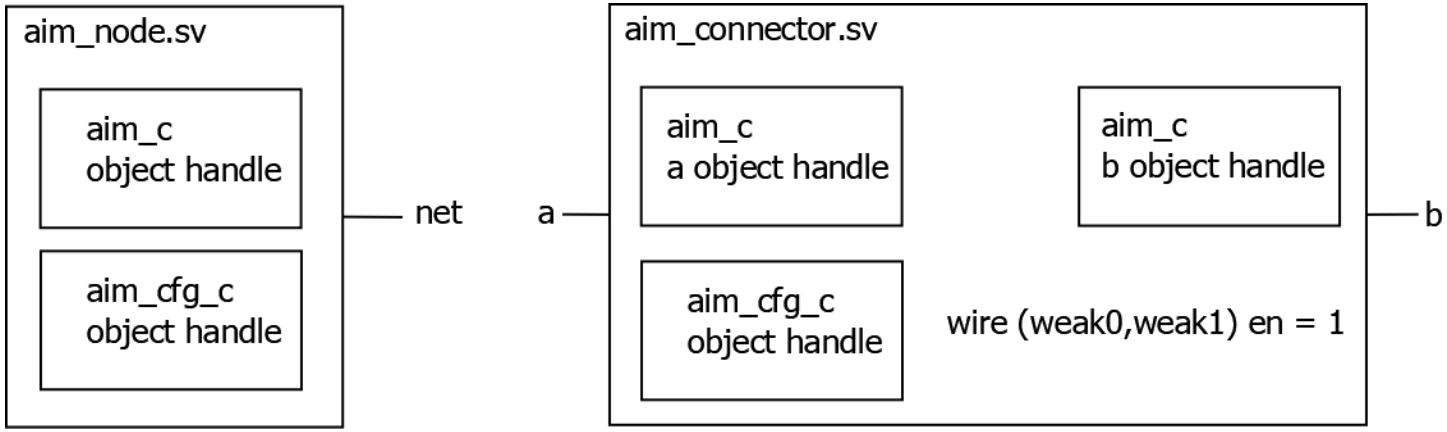
AIM System Usage




```
module supply_pad import UPF::*;
  ( input wire en, output UPF::supply_net_type vout);
  aim_upf_out_node vout_if (.net(vout));
  always @*
    if (en) vout_if.set_voltage(1.8);
    else    vout_if.set_voltage(0.0);
endmodule

module vref import UPF::*;
  ( input wire en, output wire vref_out);
  aim_node vref_out_if (.net(vref_out));
  always @*
    if (en) vref_out_if.set_voltage(0.8);
    else    vref_out_if.set_voltage(0.0);
endmodule

module mux import UPF::*;
  ( input wire  sel,
    input       UPF::supply_net_type vdd,
    input wire  vref,
    output wire dac_vref
    );
  aim_upf_in_connector (.a (vdd), .b (dac_vref));
  aim_connector (.a (vref), .b (dac_vref));
  always @* begin
    aim_upf_in_connector.en = ~sel;
    aim_upf_in_connector.en = sel;
  end
endmodule

module dac import UPF::*;
  (input wire  en,
   input       UPF::supply_net_type vdd,
   input wire  dac_vref,
   output wire dac_out);
  aim_upf_in_node vdd_if (.net(vdd));
  aim_node dac_vref_if (.net(dac_vref));
  aim_node dac_out_if (.net(dac_out));
  always @*
    if (en & (vdd_if.voltage > 1.0) & (dac_vref_if.voltage > 0.75))
         dac_out_if.set_voltage(0.8); // Drive the DAC output
    else dac_ou_ift.set_voltage(0.0);
endmodule
```

Under the Hood

- AIM nodes and connectors contain AIM objects and an AIM configuration object
- The `aim_nodes` array used to register AIM class objects on creation using their `%m` path
- AIM configuration object stores valid range properties
  - `check_ok` method is called at end of value resolution

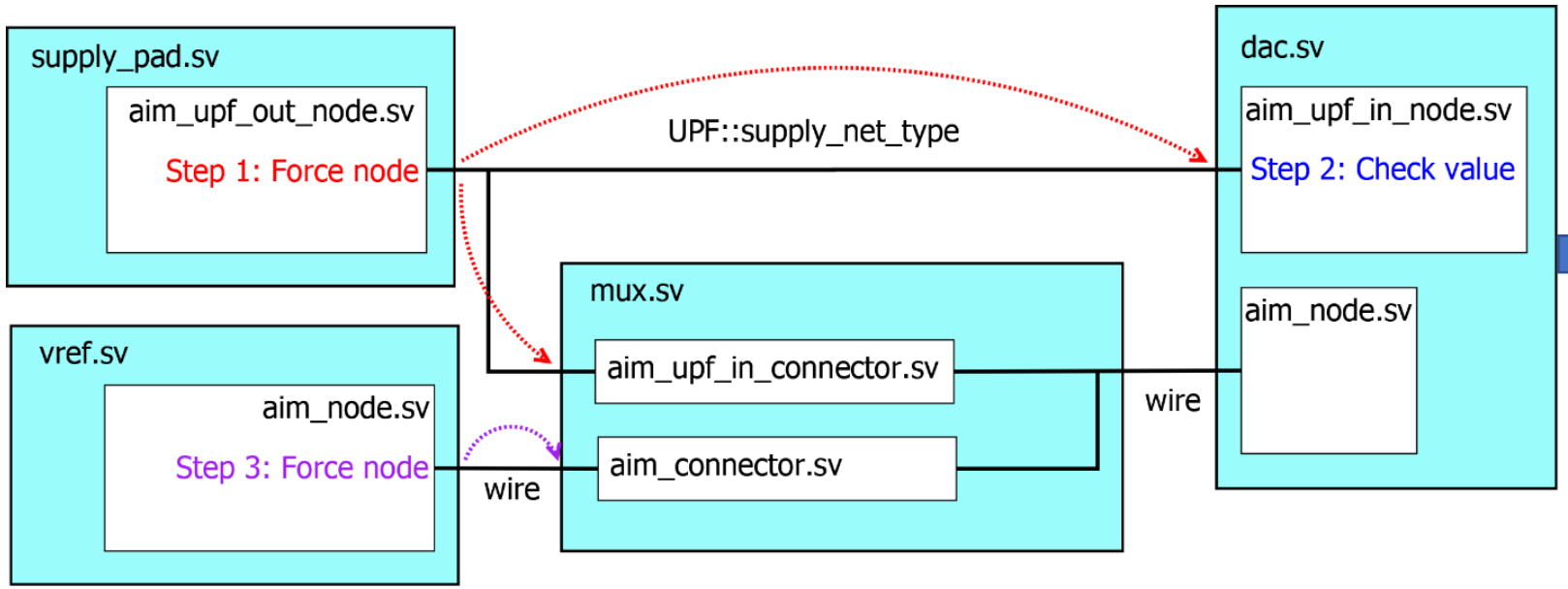



```
class aim_c;
  static aim_c aim_nodes[int][string];
  function new(string node_name);
    this.node_name          = node_name;
    aim_nodes[0][node_name] = this;
  endfunction
endclass
```

```
class aim_cfg_c;
  real min_voltage, max_voltage;
  real min_current, max_current;
  real min_resistance, max_resistance;
endclass
```

```
// Inside the aim_node module
aim_cfg_c cfg = new($sformatf("%m"));
function automatic bit check_ok();
  check_ok = 0;
  if (!(voltage inside {[cfg.min_voltage:cfg.max_voltage]})) return 1;
  if (!(current inside {[cfg.min_current:cfg.max_current]})) return 1;
endfunction

// Inside of a model using the node 'is_ok' properties to enable the block
assign model_en = enable & vdd_node.is_ok & vss_node.is_ok & vref_node.is_ok;
```

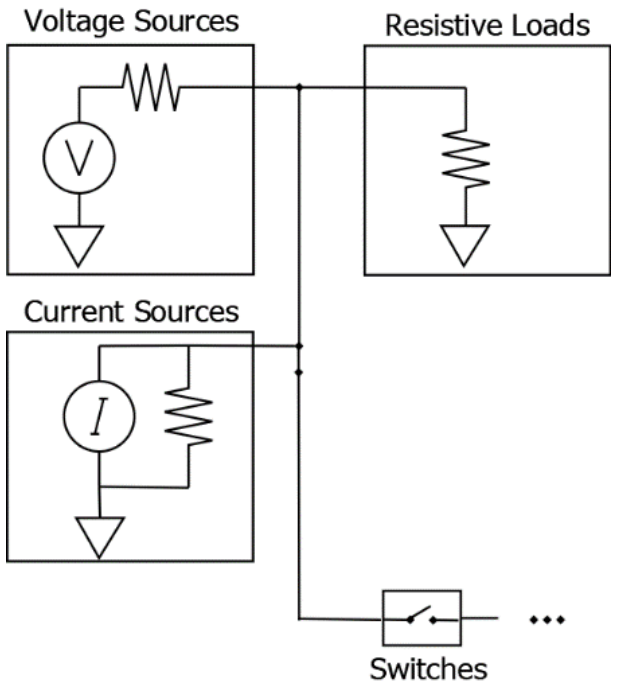
AIM Initialization

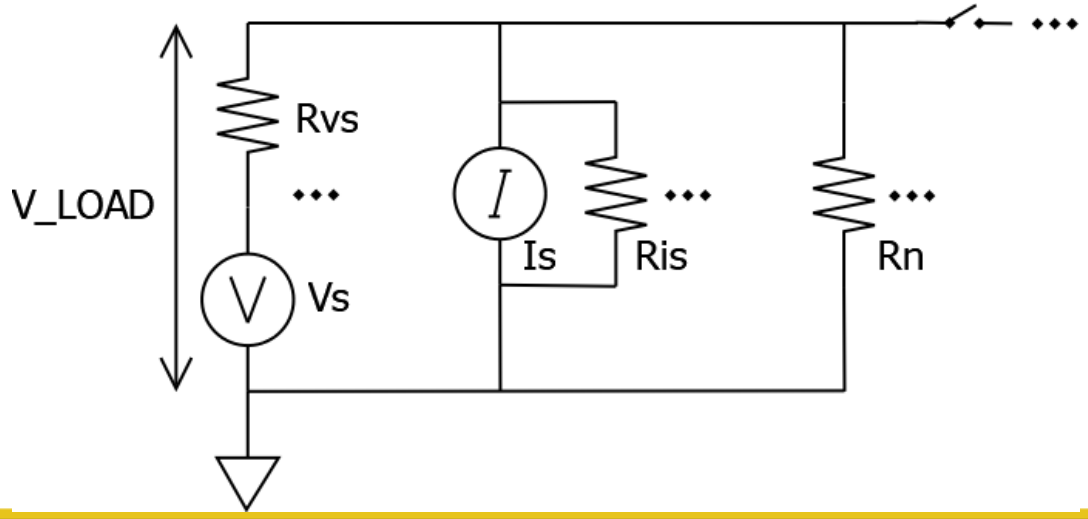



```
aim_nodes
        [0]
           ["top.supply_pad_0.vout"]
           ["top.dac_0.vdd"]
           ["top.mux_0.vdd"]
           ["top.vref_0.vref_out"]
           ["top.mux_0.vref"]
           ["top.mux_0.dac_vref"]
           ["top.dac_0.dac_vref"]
           ["top.dac_0.dac_out"]

        [1]
           ["top.supply_pad_0.vout"]
           ["top.dac_0.vdd"]
           ["top.mux_0.vdd"]
        [2]
           ["top.vref_0.vref_out"]
           ["top.mux_0.vref"]
        [3]
           ["top.mux_0.dac_vref"]
           ["top.dac_0.dac_vref"]
        [4]
           ["top.dac_0.dac_out"]
```

Value Resolution

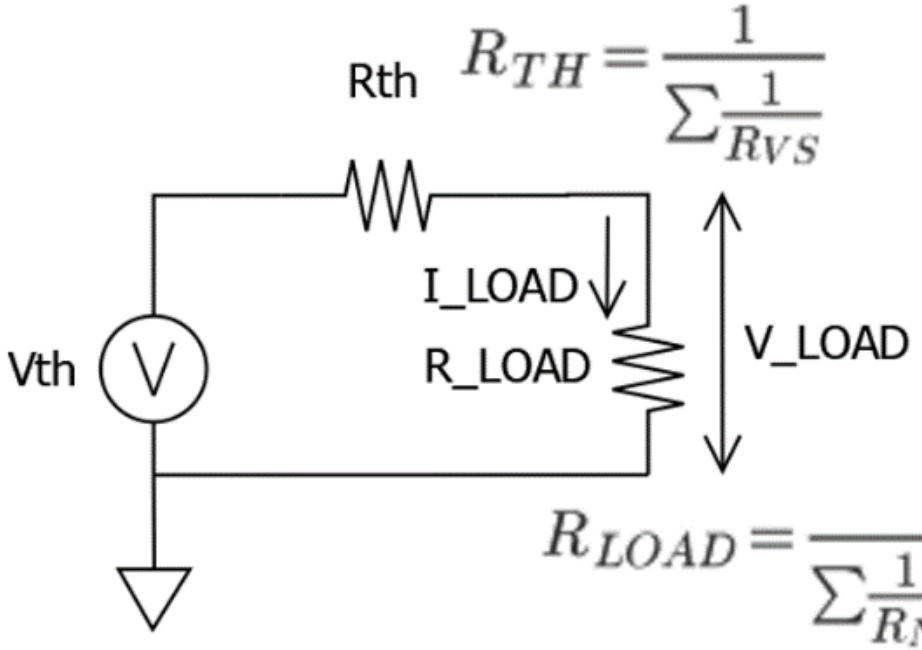
- Our systems always contained a common set of node types
  - Voltage/current sources, resistive loads, and switches
  - These defined as V_SOURCE, I_SOURCE, LOAD, and SWITCH respectively
- All nodes modeled in parallel sharing common ground





Value Resolution

- All switches resolved
- Utilize Millman's theorem to simplify and solve load voltage
  - Converts parallel network to Thevenin equivalent voltage source with load resistance



$$R_{TH} = \frac{1}{\sum \frac{1}{R_{VS}}}$$

$$R_{LOAD} = \frac{1}{\sum \frac{1}{R_N} + \sum \frac{1}{R_{IS}}}$$

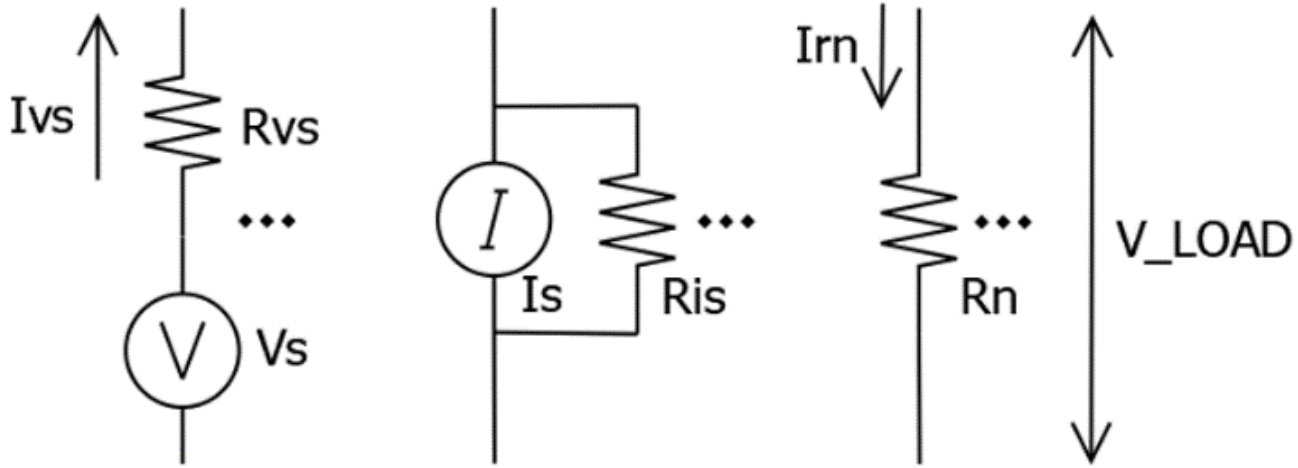
$$V_{TH} = \frac{\sum \frac{V_S}{R_{VS}} + \sum I_S}{\sum \frac{1}{R_{VS}}}$$

$$I_{LOAD} = \frac{V_{TH}}{R_{LOAD} + R_{TH}}$$

$$V_{LOAD} = I_{LOAD} R_{LOAD}$$

Value Resolution

- Use load voltage and resistance to solve branch currents

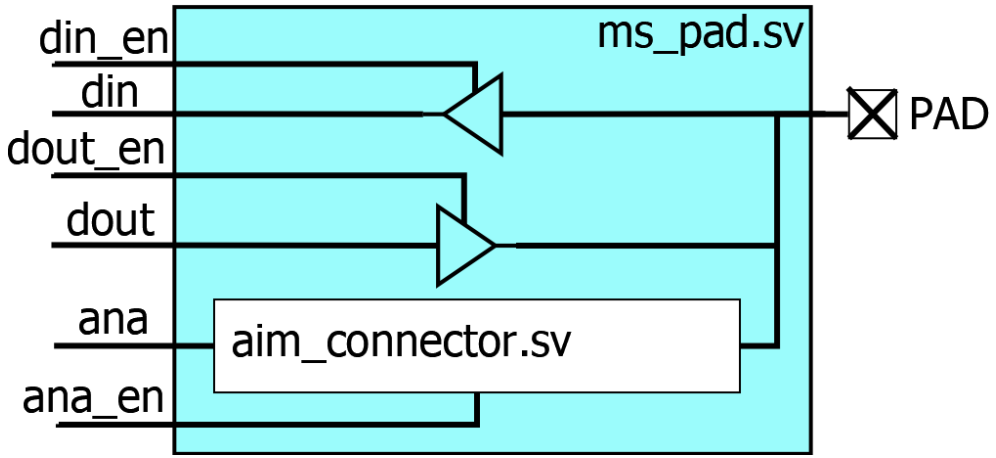


$$I_{VS} = \frac{V_{LOAD} - V_S}{R_{VS}}$$

$$I_{RN} = \frac{V_{LOAD}}{R_N}$$

Dealing with Mixed Signal Ports

- Typically encountered with mixed signal pads
- An AIM node is connected to the PAD port in the testbench to handled analog traffic
- Digital signals connect to the PAD wire directly
  - Timing annotation using specify blocks works without modification



Enhanced UPF Simulation

- Value resolution is handled by AIM
  - Input UPF ports use state and voltage in the resolution
  - Output UPF ports are driven with the resolved value
- UPF net connections through AIM provide current information
- AIM supports digital load modeling

dig.sv

```
always @(en)
  if (!en)
    // in reset
  else
    // running
```

dig_load.sv

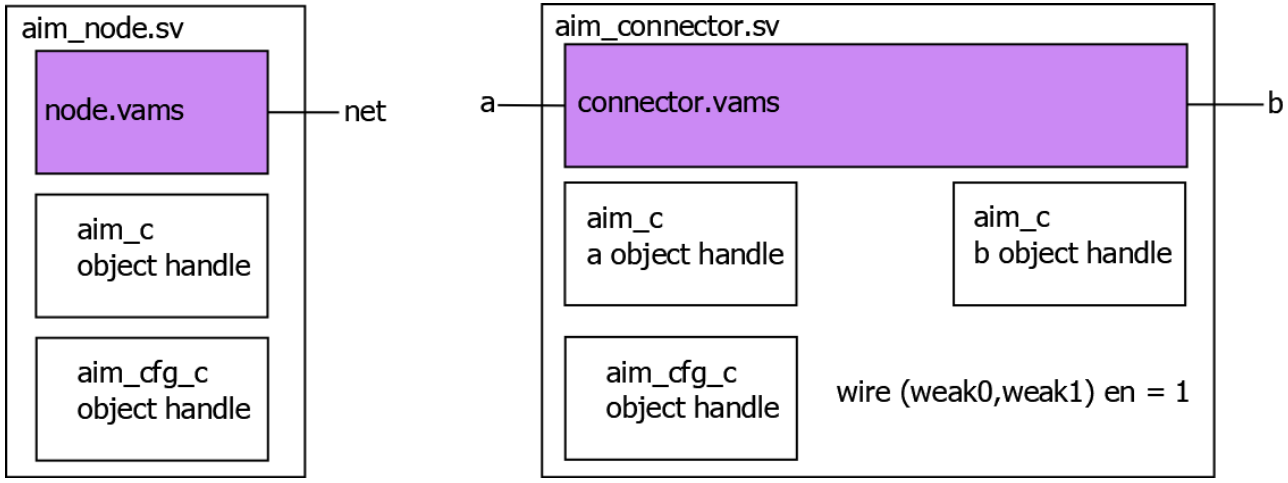
```
UPF::supply_net_type vdd;
aim_upf_in_node node(vdd);
always @(dig.en)
  if (!en) node.resistance = 10e6;
  else     node.resistance = 1e6;

bind dig dig_load load();
```

top.upf

```
connect_supply_net VDD -ports top/dig_0/load/vdd
connect_supply_net VDD -ports top/ana_0/vreg/vdd_out
```

AMS Support

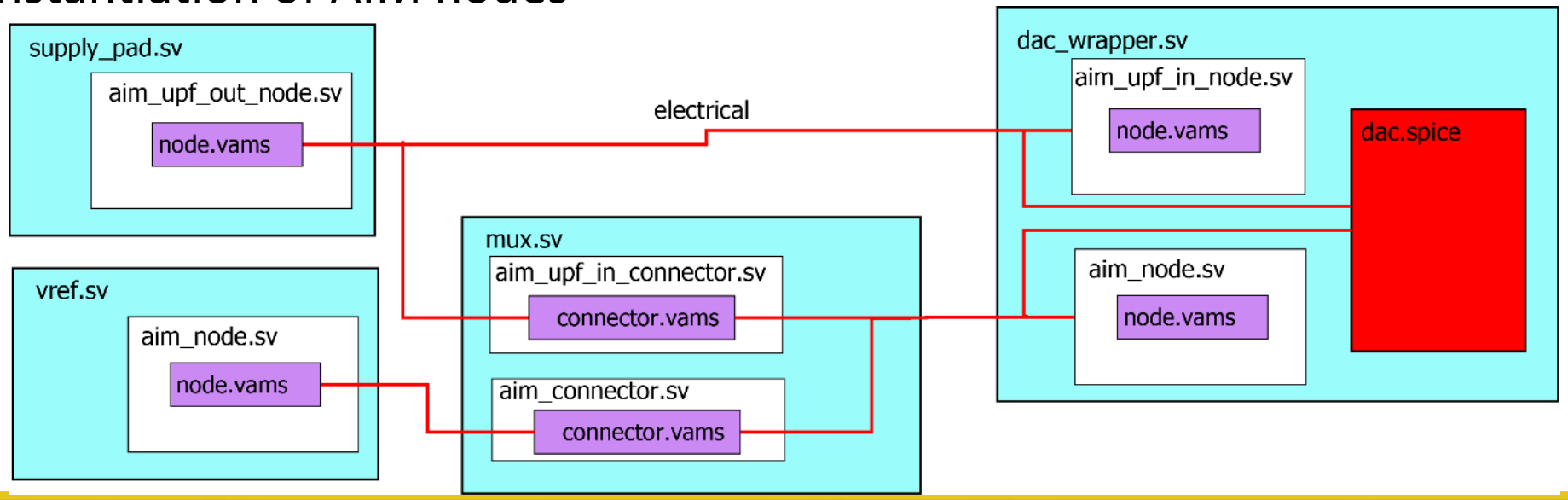
- During AMS, AIM nodes and connectors instantiate a node and connector V-AMS instance
- The AIM resolution object only updates voltage, current, and resistance properties
  - Utilizes the cosimulation engine to perform value resolution
- V-AMS instances model the node_type in an analog block




```
// 'select' assigned from the aim_node.sv module and mapped
// to the node_type enum property
wire [1:0] select;
analog begin
  case (select)
    2'b00: V(hdl_con) <+ I(hdl_con)*r_tran + voltage; // Voltage Source
    2'b01: I(hdl_con) <+ V(hdl_con)/r_tran + current; // Current Source
    2'b10: V(hdl_con) <+ I(hdl_con)*r_tran;           // Load
    2'b11: I(hdl_con) <+ 0;                           // Switch
  endcase
end
```

AMS Example

- Existing designs mostly stay the same
  - AIM nodes and connectors enable instantiation of V-AMS instances with `` `ifdef `` statements
  - The only difference is spice blocks must be in a wrapper to enable instantiation of AIM nodes



Summary

- Useful level of abstraction for modeling analog behavior
- VIR value resolution class
- Enhanced power aware simulation capabilities
- Plug-n-play cosimulation support
- Questions?