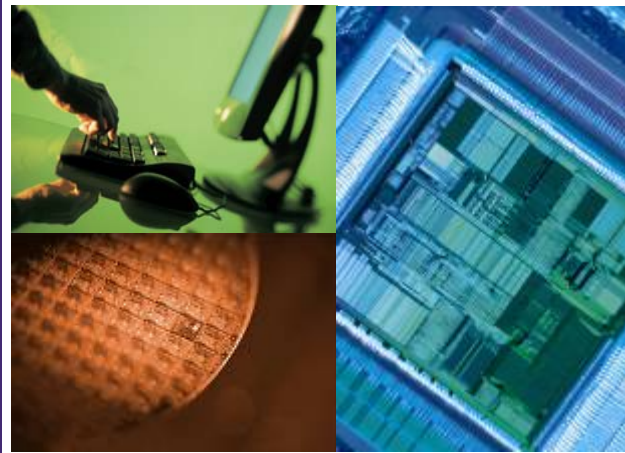


Liberty Update

Rajesh Kumar
CAE
Synopsys Inc.



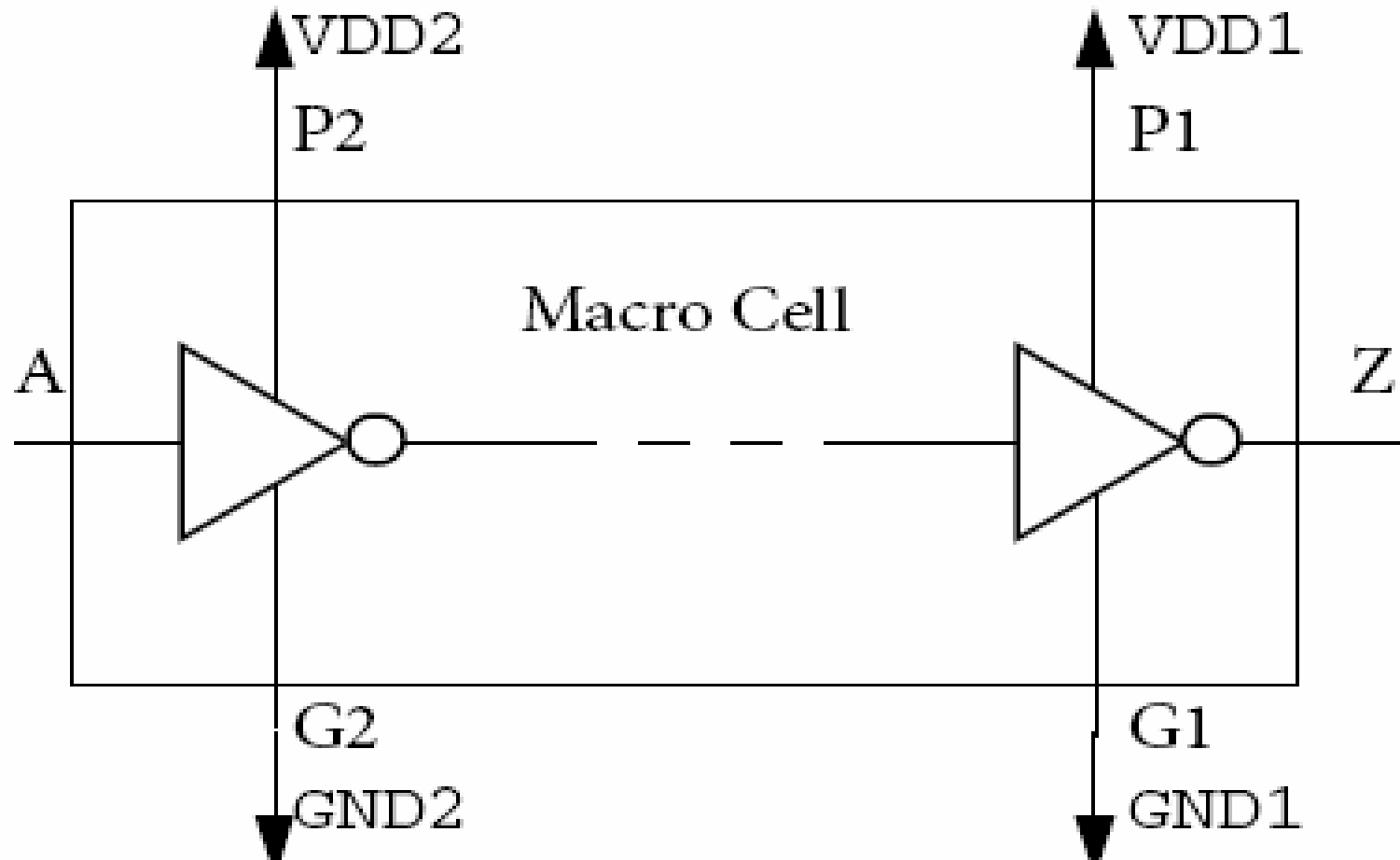
AGENDA

- **PG-pin**
- **CCS Noise**
- **CCS Power**
- **Setup/Hold Interdependence**

PG-Pin Introduction

- The **PG (power/ground) pin** support in Liberty is to provide support for real PG library pins where:
 - power pin means current source pin and
 - ground pin means current sink pin
- This new syntax will replace the old PG-pin Liberty syntax going forward

A sample Level-Shifter cell with PG-pins



PG-Pin Syntax

- **voltage_map** (library level complex attribute) These defined <voltage_name>'s are referenced by the pg_pin groups defined at the cell level

- **voltage_map**(VDD1, 3.0);

- **pg_pin** group (cell level group)

- **pg_pin**(P1) {
 voltage_name : VDD1;
 pg_type : *primary_power*;
}

Valid values of the *pg_type* attribute are : *primary_power*,
primary_ground, *backup_power*, *backup_ground*, *internal_power*
and *internal_ground*

PG-pin syntax

- **related_power_pin/related_ground_pin** pin level attributes are used to associate the PG pins to the signal pins of the cell
- **related_pg_pin** is used to associate the PG-pin with the leakage and the internal power tables
- **output_signal_swing** and **input_sinal_swing** groups with attributes **low** and **high** will be used to define the signal swings on the output and input signal pins respectively

PG-Pin syntax example

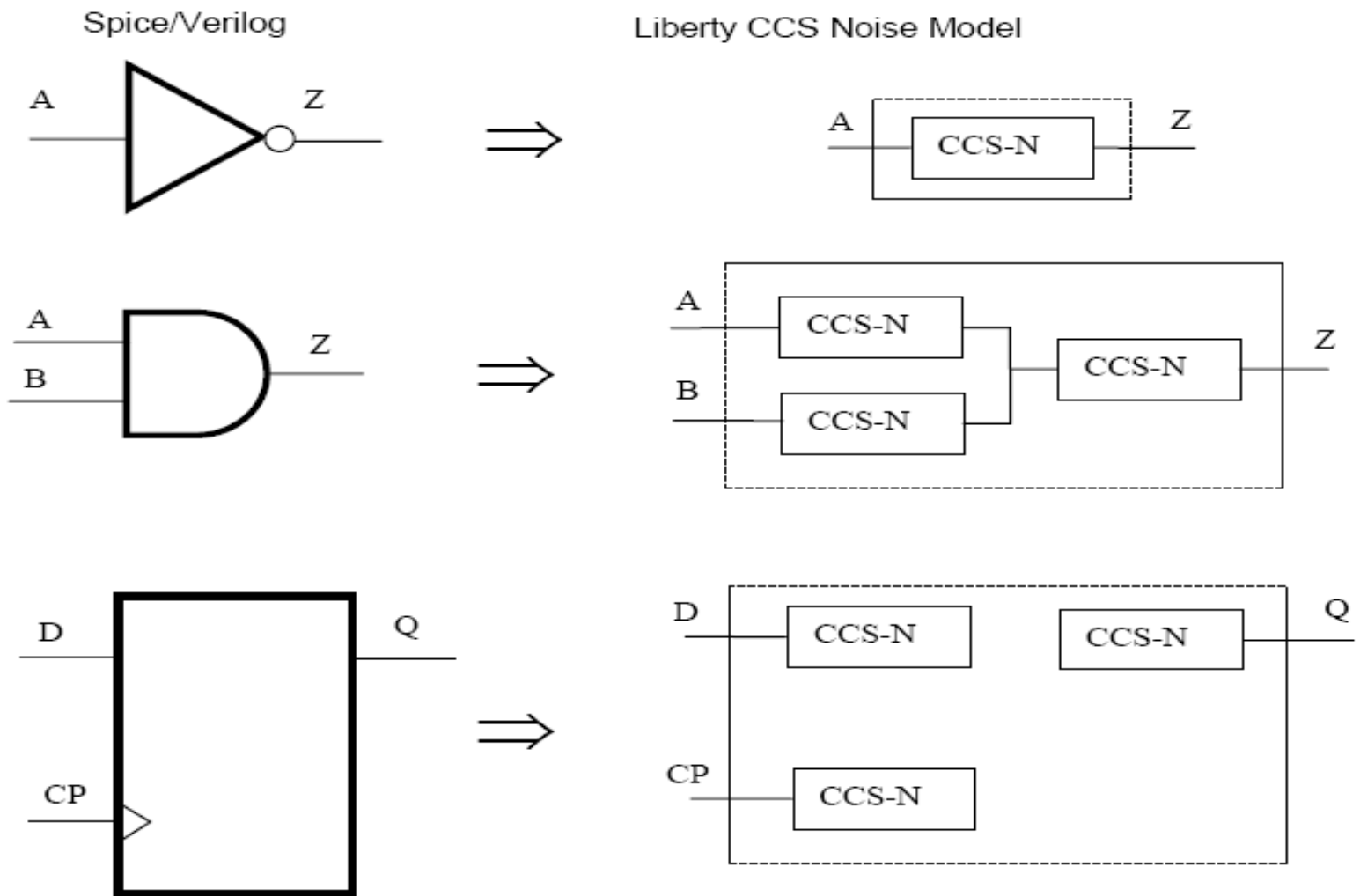
```
library(sample) {  
  voltage_map(VDD1, 3.0);  
  voltage_map(VDD2, 3.1);  
  voltage_map(GND1, 0.3);  
  voltage_map(GND2, 0.0);  
  ...  
  cell(test) {  
    pg_pin(P1) {  
      voltage_name : VDD1;  
      pg_type : primary_power;  
    }  
    ...  
    pg_pin(G2) {  
      voltage_name : VDD2;  
      pg_type : primary_ground;  
    }  
    ...  
    leakage_power() {  
      when : "!A";  
      value : 1.5;  
      related_pg_pin : P1;
```

```
    pin(A) {  
      direction : input;  
      related_power_pin : P2;  
      related_ground_pin : G2;  
      input_signal_swing() {  
        low : 2.0;  
        high : 2.8;  
      }  
      ...  
    }  
    pin(Z) {  
      direction : output;  
      related_power_pin : P1;  
      related_ground_pin : G1;  
      output_signal_swing() {  
        low : 2.1;  
        high : 2.95;  
      }  
      timing() {  
        ...
```

CCS Noise Support

- The CCS Noise is a set of characterization data that provides sufficient information for noise failure detection on cell inputs, calculation of noise bumps on cell outputs, and noise propagation through the cell
- The CCS Noise data includes:
 - several Channel Connected Block(CCB) parameters
 - DC current tables
 - timing tables for rising and falling transitions
 - timing tables for low and high propagated noise

Channel Connected Block (CCB)



CCS Noise Syntax

- The **ccsn_first_stage/ccsn_last_stage** groups are defined inside the timing group or the pin group, and they are used to specify the CCS Noise data of the first stage or the last stage of the CCB
- A **ccsn_first_stage** or **ccsn_last_stage** group contains the following information:
 - a set of CCB parameters:
 - **is_needed** simple attribute
 - **is_inverting** simple attribute
 - **stage_type** simple attribute
 - **miller_cap_rise /miller_cap_fall** simple attribute
 - a two-dimensional dc current table : **dc_current** group
 - two timing tables for rising and falling transitions :
output_current_rise group **output_current_fall** group
 - two noise tables for low and high propagated noises :
propagated_noise_low group **propagated_noise_high** group

CCS Noise Sample library

```
library(sample) {  
  ...  
  lu_table_template(ccsn_dc) {  
    variable_1 : input_voltage;  
    variable_2 : output_voltage;  
  }  
  lu_table_template(ccsn_timing_lut) {  
    variable_1 : input_net_transition;  
    variable_2 : total_output_net_capacitance;  
    variable_3 : time;  
  }  
  lu_table_template(ccsn_prop_lut) {  
    variable_1 : input_noise_height;  
    variable_2 : input_noise_width;  
    variable_3 : total_output_net_capacitance;  
    variable_4 : time;  
  }  
  ...  
  
  cell(INV) {  
    ...  
    Pin(A) { ... }  
    Pin(Y) {  
    ...  
  }  
}
```

```
ccsn_first_stage ( ) {  
  is_needed : true;  
  is_inverting : true;  
  stage_type : both;  
  miller_cap_rise : 0.00055;  
  miller_cap_fall : 0.00084;  
  dc_current (ccsn_dc) { ... }  
  output_voltage_rise ( ) {  
    vector (ccsn_timing_lut) { ... }  
  }  
  output_voltage_fall ( ) {  
    vector (ccsn_timing_lut) { ... }  
  }  
  propagated_noise_low ( ) {  
    vector (ccsn_prop_lut) { ... }  
  }  
  propagated_noise_high ( ) {  
    vector (ccsn_prop_lut) { ... }  
  }  
}
```

CCS Power

- CCS power represents the different component of power consumed in a cell like leakage power and dynamic power using current waveforms

- **Leakage Current Syntax**

Since CCS power is current-based data, leakage current on the PG pin is captured instead of leakage power as specified in NLPM format

Example:

```
cell(<cell_name>) {  
  ...  
  leakage_current() {  
    when : <boolean expression>;  
    pg_current(<pg_pin_name>) {  
      value : <float>;  
    }  
  }  
  ...  
  leakage_current() {                               /*without when statement */  
  ... }  
}
```

CCS Power

- **Dynamic Power**

- Instantaneous power data (current waveforms) on the PG pin is captured. The table is depended on transition time of a toggling input and capacitance of the toggling outputs.

```
pg_current_template ( CCS_power )
```

```
{  
  variable_1 : input_net_transition ;  
  variable_2 :  
total_output_net_capacitance ;  
  variable_3 : time ;  
}
```

```
...
```

```
  cell ( XYZ ) {
```

```
    ...
```

```
    dynamic_current() {
```

```
      when: "D";  
      related_inputs : "CP";  
      related_outputs : "Q";
```

```
switching_group ( ) {
```

```
  input_switching_condition(rise);
```

```
  output_switching_condition(rise);
```

```
  pg_current (VDD ) {
```

```
    vector ( CCS_power ) {
```

```
      reference_time : 0.01;
```

```
      index_1 ( "0.01" )
```

```
      index_2 ( "1.0" )
```

```
      index_3 ( "0.000, 0.0873, 0.135, 0.764" )
```

```
      values ( "0.002, 0.009, 0.134, 0.546" )
```

```
    }
```

```
  ...
```

Setup/Hold interdependence

- The Liberty format has been enhanced to allow identification of interdependent pairs of setup/hold constraint tables.
- The library APIs for access to interdependent pairs of setup/hold constraint tables must be distinct from APIs for the existing non-interdependent setup and hold constraint tables.
- Liberty will support this enhancement by introducing a simple attribute in timing arc to distinguish setup/hold interdependence data from original setup/hold constraints.

Setup/Hold interdependence

Example:

```
/* setup/hold constraint timing  
blocks without the  
interdependence_id */
```

```
timing () {  
    related_pin : "CLK";  
    timing_type : setup_rising;  
    interdependence_id : 1;  
    rise_constraint(cons) {  
        ...  
    }  
    fall_constraint(cons) {  
        ...  
    }  
}
```

```
timing () {  
    related_pin : "CLK";  
    timing_type : hold_rising;  
    interdependence_id : 1;  
    rise_constraint(cons) {  
        ...  
    }  
    fall_constraint(cons) {  
        ...  
    }  
}
```

SYNOPSYS®

Predictable Success