Vectorless Dynamic Analysis

RedHawk-SC Modularized Training Series

2020_R3

Ansys

Info For Attendees

- Please join audio via Audio Broadcast option
- Please use the Q&A window to clear queries and one of the panelists will answer it.
  - Direct questions to all of panelists
- This training is for 3 hours.
  - Will break into 3 sessions of 50 mins each, with 10 mins Q&A at the end of all three.
- The slides and recording will be available at Ansys website within a week
  - Registered participants will be receiving emails with the link
- For offline follow up of queries, please reach out to your local AE or email eldo.baby@ansys.com

People on Panel

- **Host**
  - Eldo N Baby – Lead Product Specialist

- **Panelists**
  - Siddalingesh Tenginakai – Lead Product Specialist
  - Ramesh Agarwal – Lead Product Specialist
  - Anusha Gummana - Lead Product Specialist
  - Sojan Philips - Lead Product Specialist
  - Sankar Ramachandran – Director Product Specialist

Prerequisites for the training

No	Training Program	Expectations – Must Know
1	Chapter 01 : Introduction_to_SeaScape	• Reading in input data, performing data integrity checks and creating base views
2	Chapter 02: Static Analysis	• Doing a static analysis

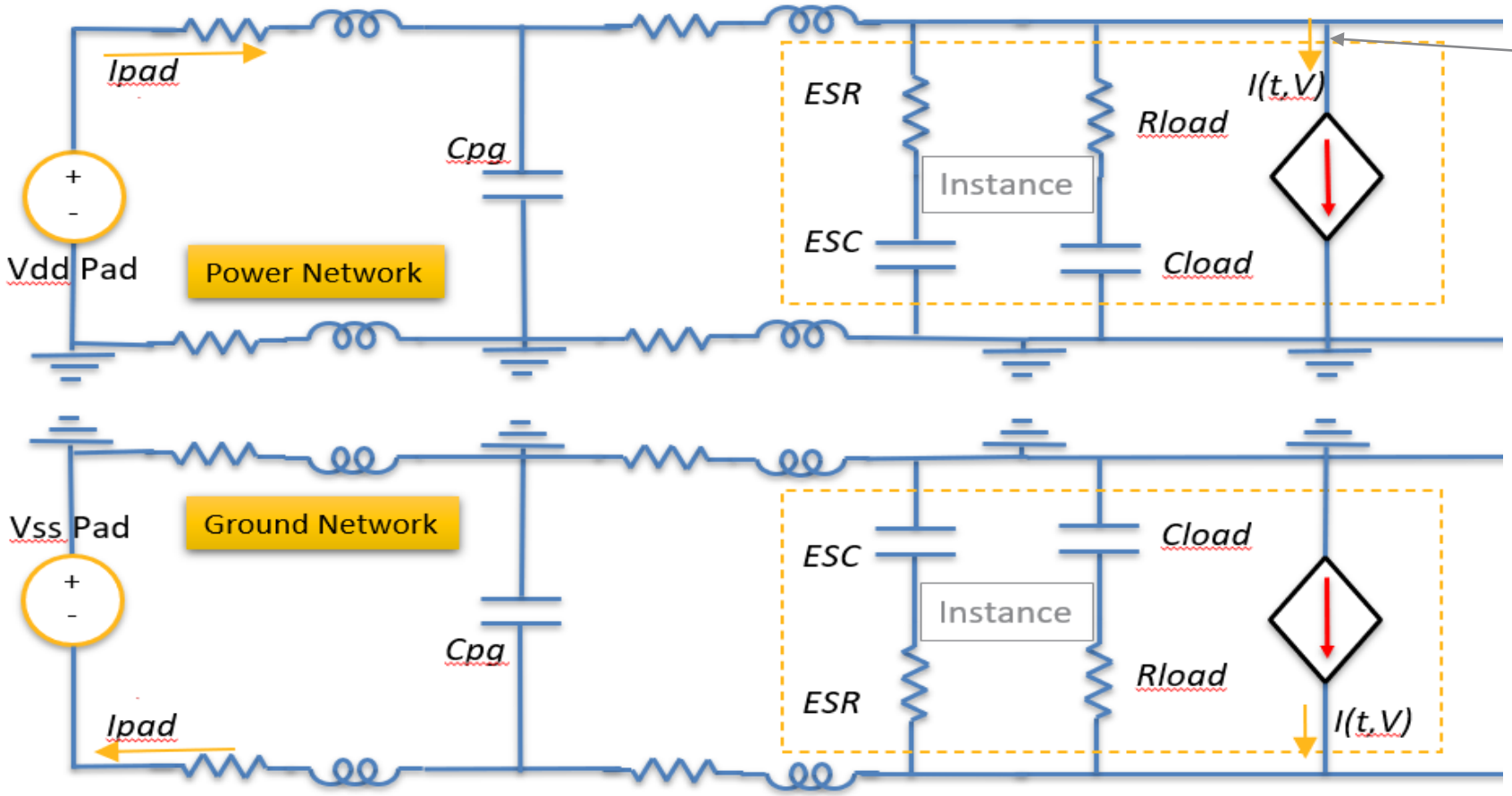
Training Agenda

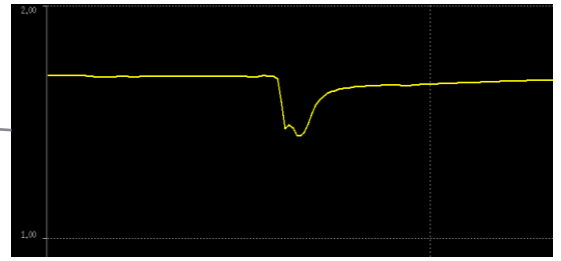
- Dynamic Voltage Drop Analysis Theory
- Vectorless Scenario
- No Propagation Vectorless (NPV) Scenario
- Logic Propagation Scenario
- Controlling Switching of Macros
- Event Replay Flow
- Training Labs

Dynamic Voltage Drop Analysis Theory

Dynamic Voltage Drop Problem Definition

- PWL current for switching instance
- From transient simulation $V_{dynamic}$ waveform probed at instance PG pins
- Intrinsic resistance ESR and Intrinsic capacitance ESC from APL or CCS-P current / cap models

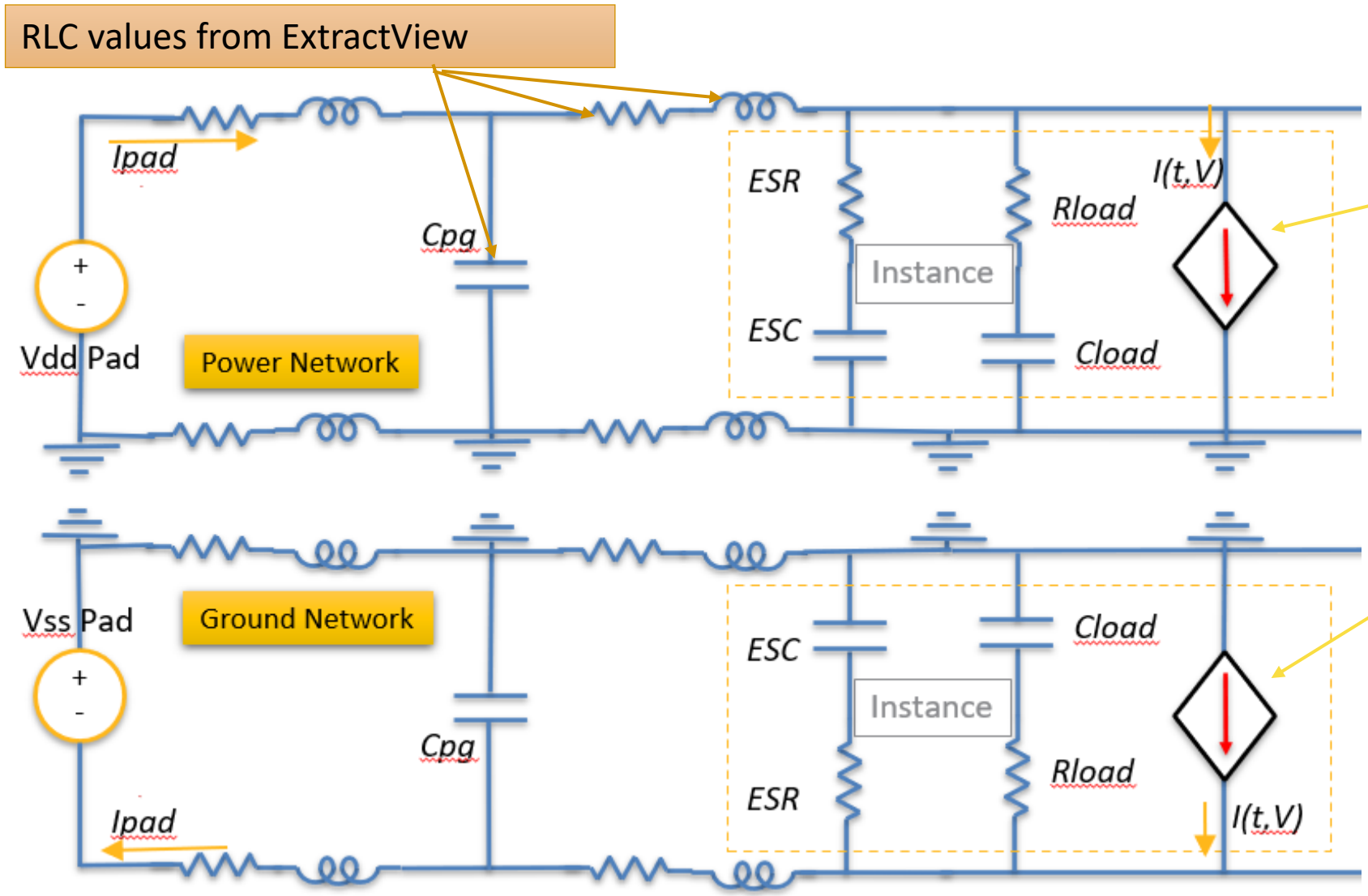


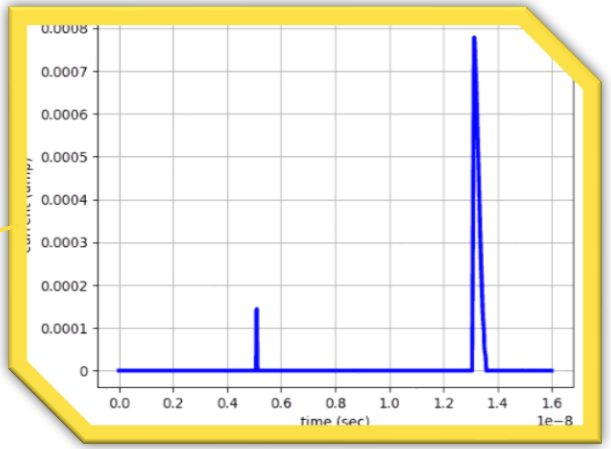


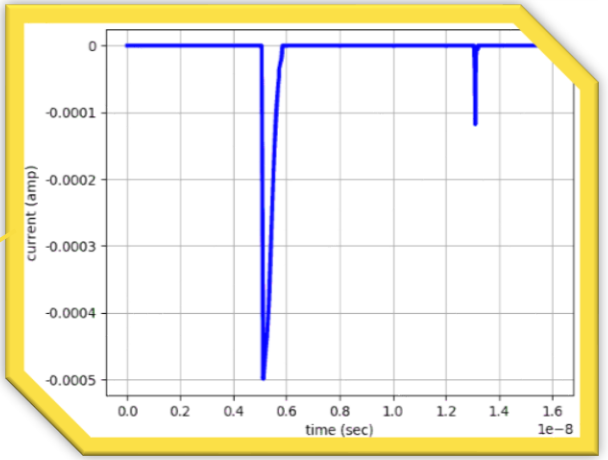
- On-chip power/ground network ➔ R,L,C mesh
- Instances ➔ i(t,V) sources, effective decap and effective series resistance

Dynamic Voltage Drop Problem Definition

RLC values from ExtractView



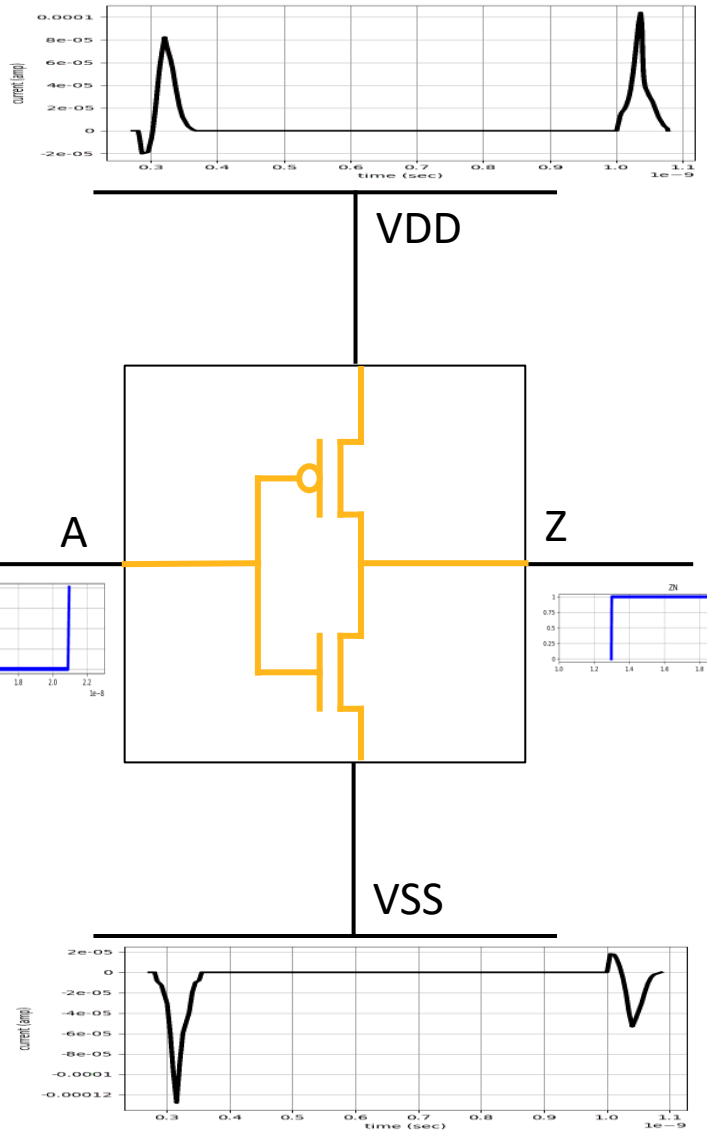




PG Currents from ScenarioView
What time and how much current

Modelling Demand Currents on PG pins

For dynamic analysis, we need currents at VDD and VSS pins



Simultaneous switching of instances in same region causes localized dvd effects

These currents are governed by logical events at input and output pin, here, A and Z

Combined total current across a region or full design results in package drop effects

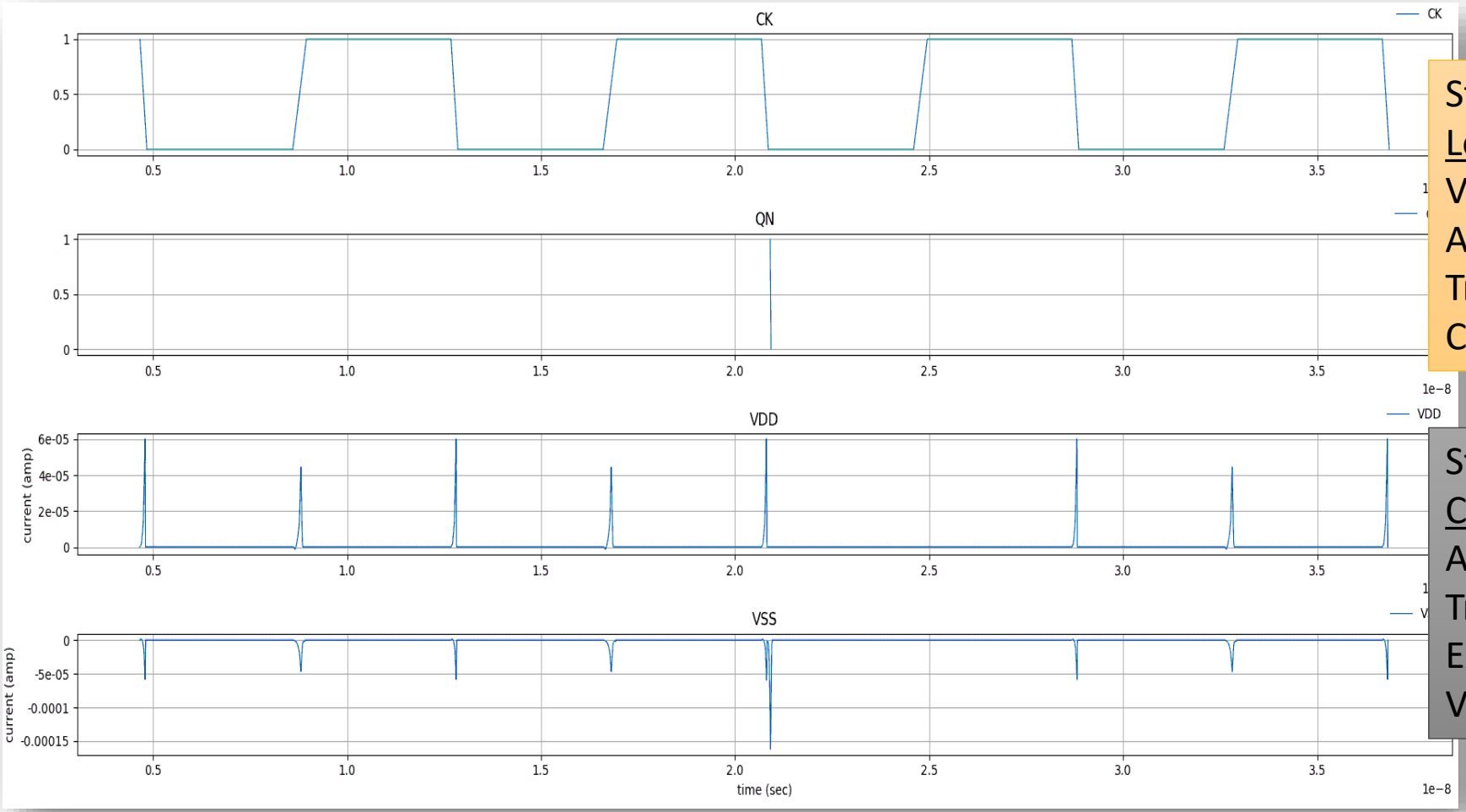
Many aspects are crucial here :

- How many events are present
- What time each of them occurs
- Transition times of each event

ScenarioView : Two Stages

- ScenarioView has two stages
  - Event Creation which gives Logical Events
  - Current Source Generation which gives Currents at PG pins
- Previous slides discussed about importance and impact of first stage of logical event creation
- Many different ways and options to configure event creation
- Creating currents at PG pins does not need much user inputs as logical event creation : just pick up currents from library data
- This training module discusses vectorless ways of generating the first stage: logical events

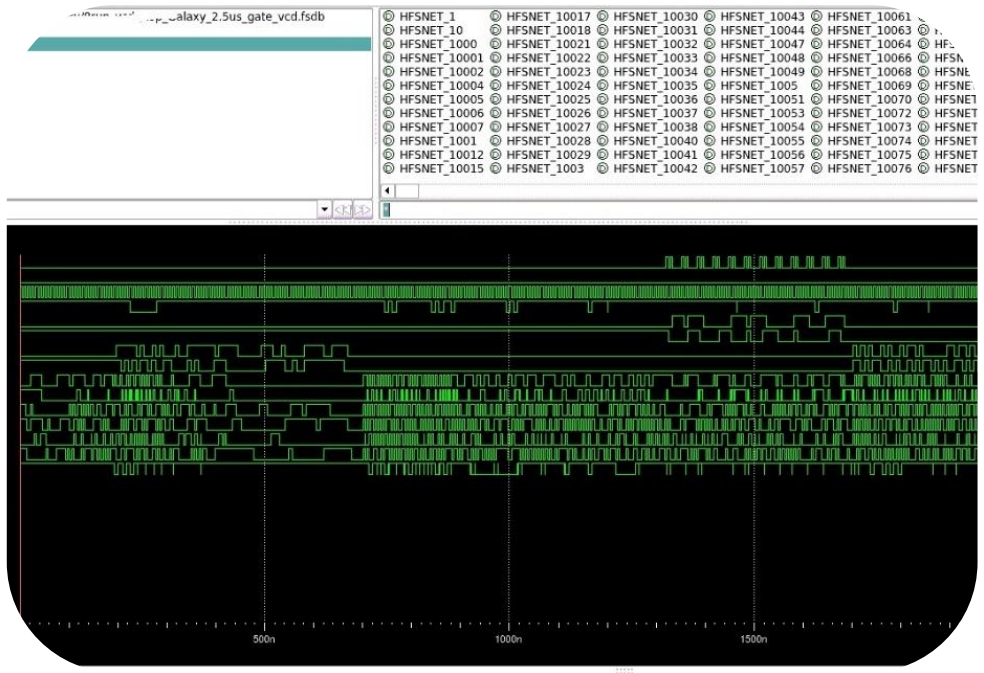
Two Stages : Example

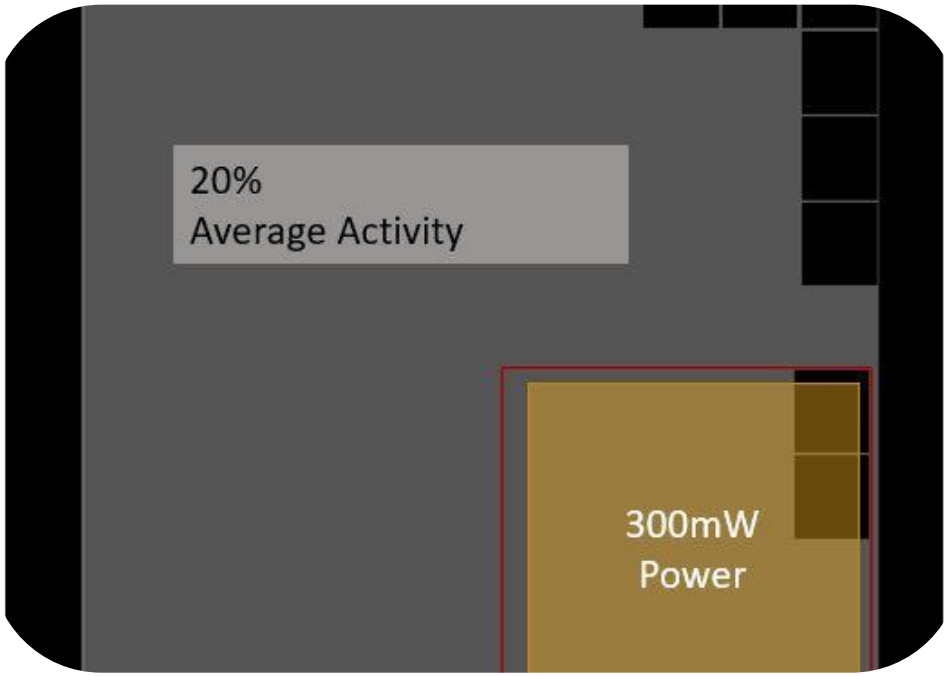


Stage 1 :
Logical Events
VCD/Vectorless Ways
Arrival Time from STA/Propagation
Transition Time from STA/Propagation
Clock Root Information

Stage 2 :
Current Waveform
APL/CCSP/NLPM Sources
Transition Time vs Load Cap Tables
Exact Arc from Logic Events
Voltage Domain Info

Inputs/Flows for Logical Events





Vector/VCD

Externally specified switching scenario

Pro : Exact Switching Information

Con : VCD not readily available

Vectorless

RedHawk-SC creates switching scenario

Pro : Setup early/easily

Con : May not reflect a possible event sequence

Vectorless Scenario

Dynamic Analysis: Challenges

Grid Complexity
- Geometries: 10B+
- Extracted circuit nodes: 10B+
- Sparse Matrix size: 10B x 10B +
- Memory needed: 2TB+
- Local + Global coupled system

Logic Complexity
- Instances: 2B+
- Flip-flops: 100M+
- Macros: 1M+
- Reachable states is infinite

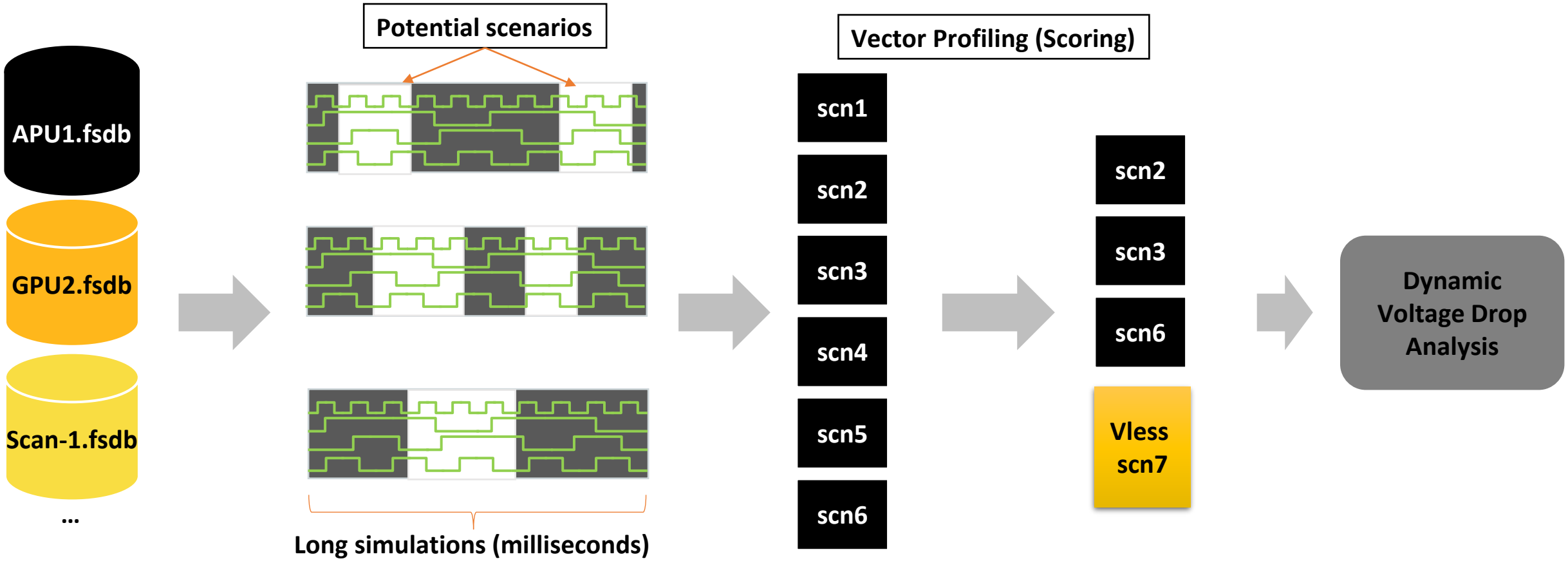
Timing Complexity
- # of timing paths: huge
- # of arrival times combination is infinite
- Each instance has a range of arrival times (from logic and PVT, SI effects)

How to get high coverage in reasonable runtime with good accuracy?
- Identify the issues that are most likely to lead to a real silicon issue
- Help to determine the root cause
- Help to repair the issue

Different Types of Dynamic Scenarios

Vector Input	• RTL VCDs/FSDB's : Some part of design populated by tool • Gate VCDs – SDF annotated and unannotated
Vectorless	• Logic Propagation based , Power Controlled (PCVS) • No Propagation Vectorless (NPV) • Activity or Power driven
Special Flows	• Scan shift modes for BIST , MBIST , ATPG test operations • Can be VCD based or Vectorless • Power Transient mode in vectorless analysis for high DI/DT scenario

Choosing Scenarios – Mix Of Vector + Vectorless Approaches



Gather simulation output from functional & test modes

Identify potential scenarios (High activity, Large changes)

Score potential scenarios individually on metrics

Select subset of scenarios for best overall coverage + Add vectorless

Importance of Vectorless Approaches

Simulation Vectors Often Insufficient

- Available too late or not available
- Functional vectors may not be best to identify voltage drop problems
- Even good vectors will be incomplete and need to be augmented for full coverage

"Synthesize" Vectors for Dynamic Analysis?

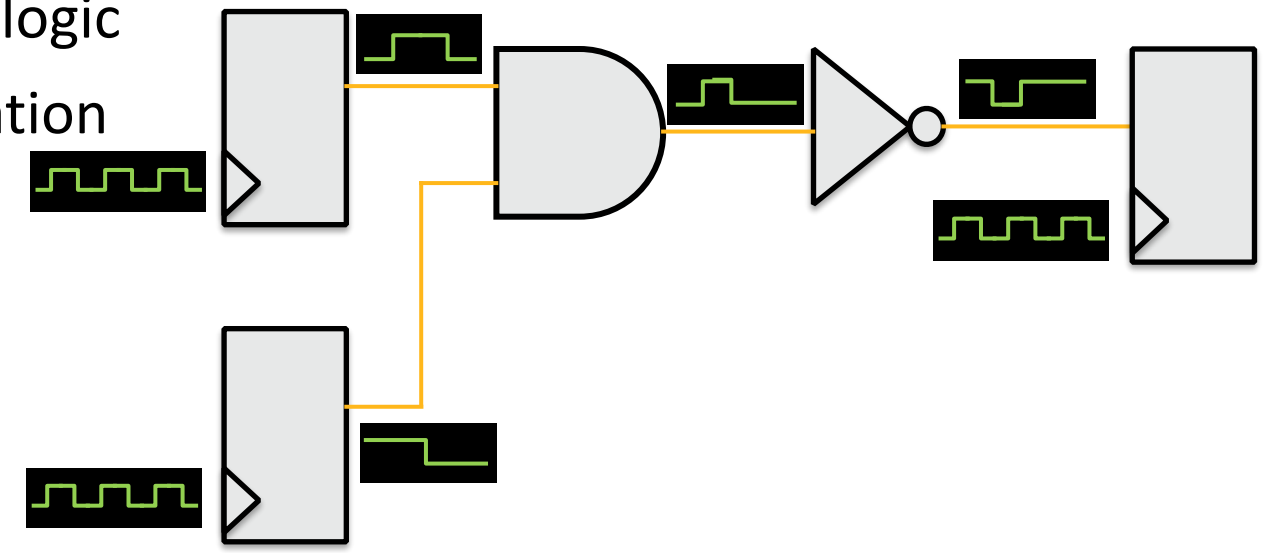
- Known as vectorless analysis since the user doesn't supply vectors
- Many methods to do vectorless

Type of Vectorless Approaches in RHSC

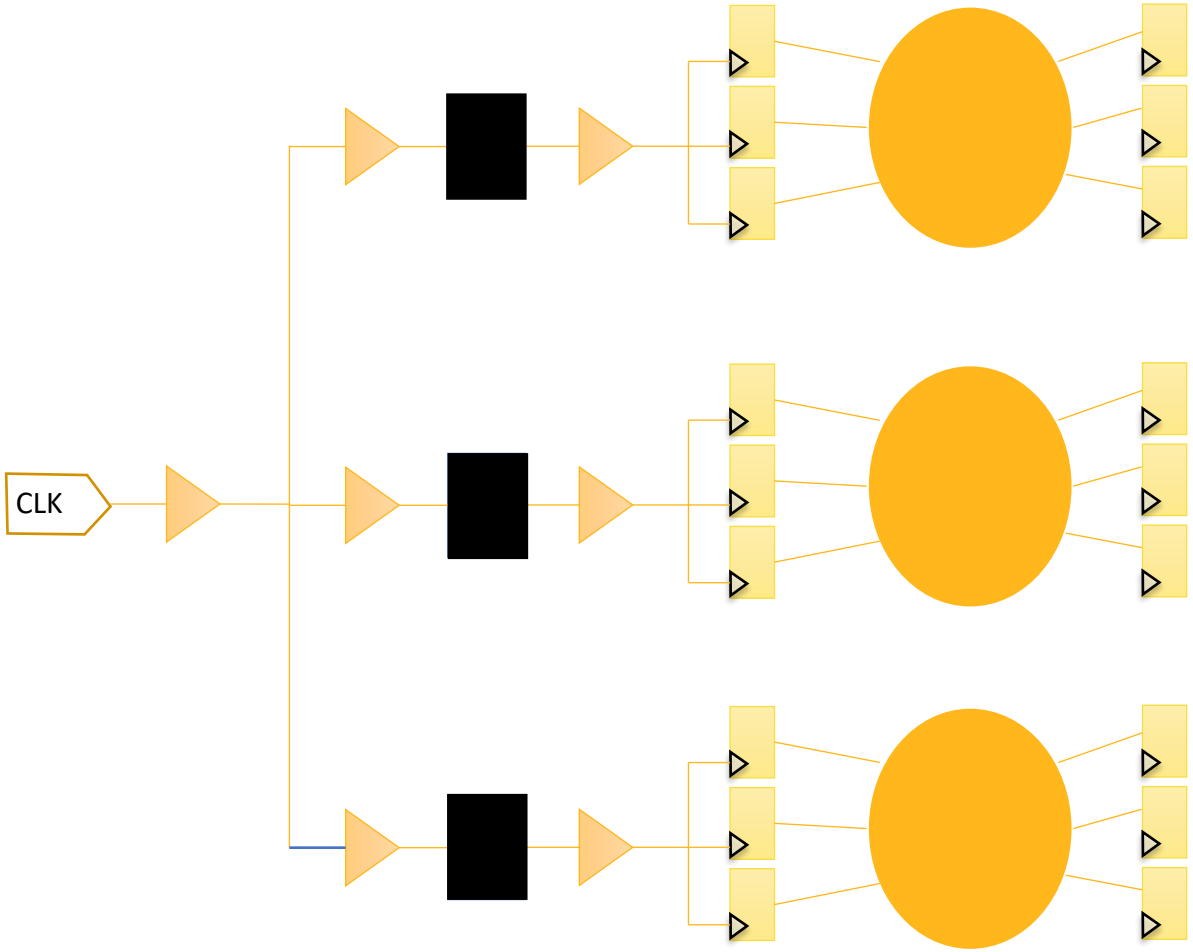
- Logic propagation based vectorless
- No Logic Propagation vectorless (NPV)

Logic or Event Propagation: Vectorless

- Waveform propagated similar to a logic simulator. Cell delay and net delay considered
- First clock network propagated , then data path
- Data paths propagated from flip-flop and macro outputs. Event there derived :
  - VCD (can be RTL or gate) data if available, or
  - Random switching based on user settings
- Events propagate through combinational logic
- Uses cell delay and logic function information
  - Net delay can also be added
  - Events more realistic
  - Considers logical and temporal correlation



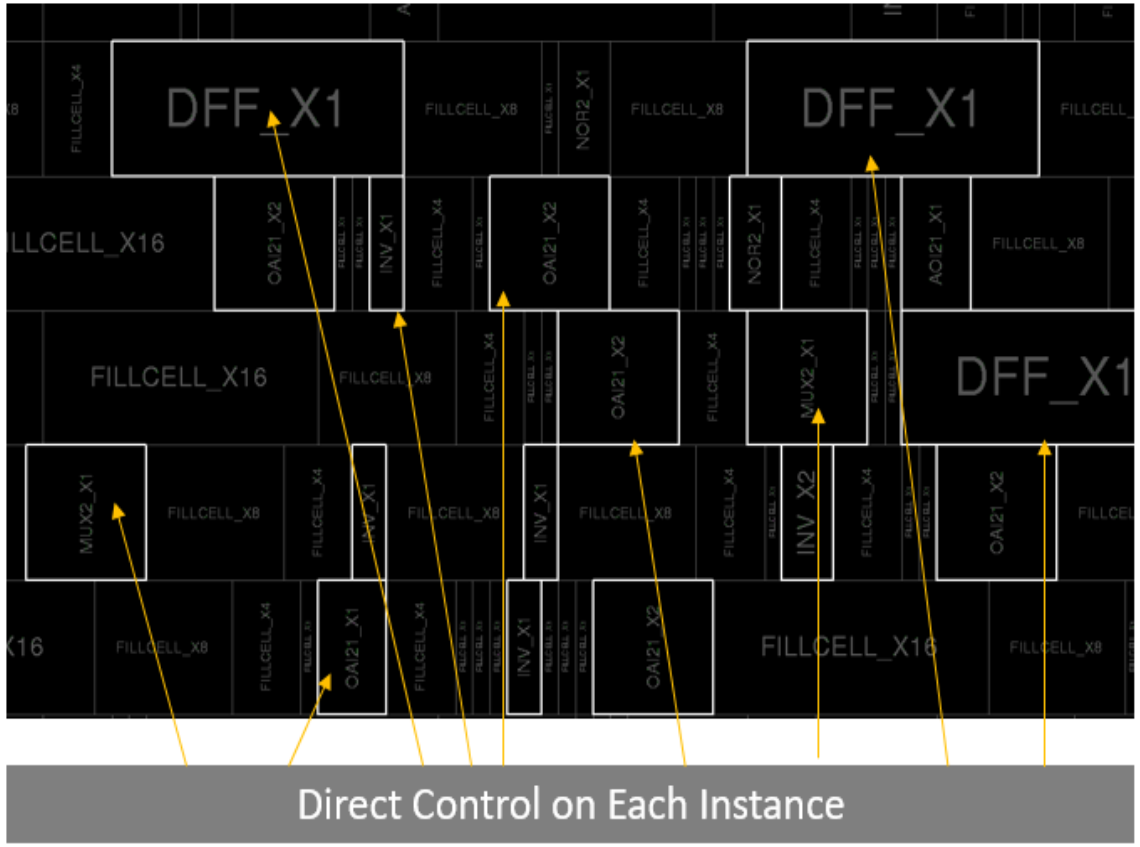
Power-Constrained Vectorless Scenario (PCVS)



- Event Propagated, logically consistent scenario
- Delay-aware propagation
- User-specified target power
  - Top or Top + Block
  - Meets power by enabling/disabling ICGs, then by flop toggle rate
  - Supports mixed flow where some blocks have gate-level VCD or FSDB
- Maximizes ICG coverage
  - Different ICGs enabled per frame

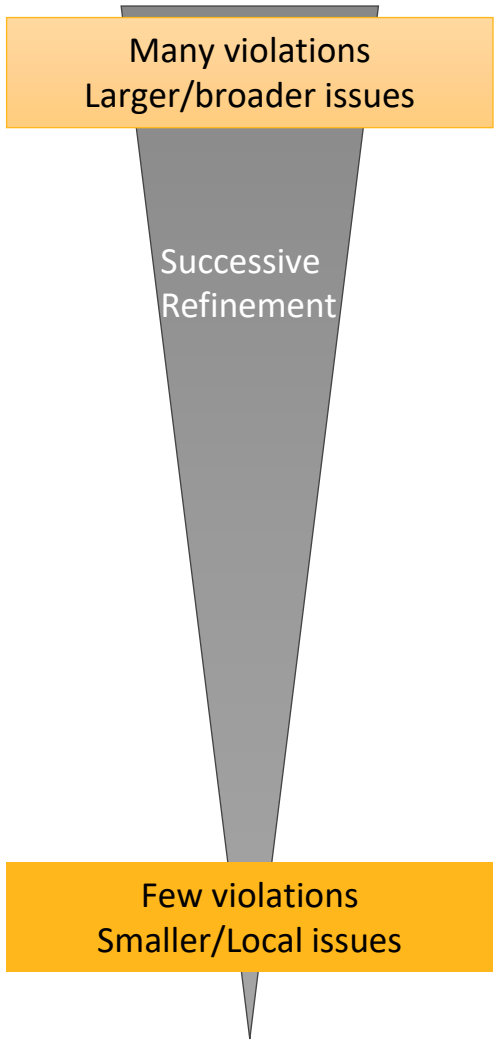
PCVS is the only vectorless engine in industry that can perform the complex task of meeting logic and power together

No-Propagation Vectorless(NPV) Scenario



- Switching of each instance directly determined, rather than propagation from flip-flops
- Highly scalable/parallel - very efficient in memory and runtime
- Very high coverage in short simulation duration
- Tight matching of power/activity targets
- Input toggle rates or target power per block

Convergent Prevention/Repair Flow for Voltage Drop



BQM: Build Quality Metric

- Measure relative weakness per PG domain across entire power grid
- Run from power planning through final implementation to ensure consistent resistance gradients as design matures

PeakTW

- Find placement issues by modeling possible peak current due to local simultaneous switching
- Run from early placement onward to identify issues as design is closed
- Merge with BQM results to determine a hazard metric for every instance PG pin

PCVS: Power-Constrained Vectorless Scenario

- Scenario for specific operating mode(s) with realistic propagated switching
- Run transient AnalysisView (typically 10 to 100ns duration each)
- Recommended for block level run's signoff
- Optional: analyze timing effect of voltage drop

No-Prop Vectorless Scenario

- Switch almost all instances in relatively few clock cycles
- Run transient AnalysisView with this scenario and fix outliers
- Excellent frequency domain power coverage for long duration simulations with full chip + package response

Recommendation : NPV for full chip, PCVS at block

- PCVS gives excellent control over average current signature while keeping logical coherence
- PCVS for full chip runs will be expensive in terms of memory and runtime, especially if big DEF blocks are involved
- Recommend to use NPV for full-chip runs, with events in blocks supplied by event replay
- Event Replay Flow explained later

Vectorless Dynamic Scenario : Two Broad Flows

No Propagation Vectorless

- Fast and Light
- Easily meets Input Target Power/ Activity

Logic Propagation Vectorless

- Logically Coherent Events
- Built in Delay and Transition Time Calculation

No Propagation Vectorless (NPV) Scenario

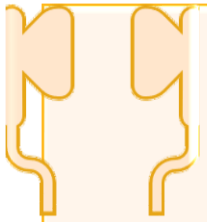
No Propagation Vectorless : NPV

- Features
- Syntax and Examples
- Concept of Frames
- Target Power Input
- Toggle Rate Input
- Annotated Instance Power Input
- Controlling Clock Frequencies and Arrival Times
- Specific Control of Switching Sequence per Instance

Features

Fast and Light

- 48M Logical Instances for 6 clock cycles in 30 mins, peak memory 4 GB

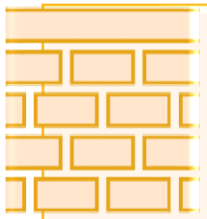


Easier Control of Switching

- Directly determine switching of each instance

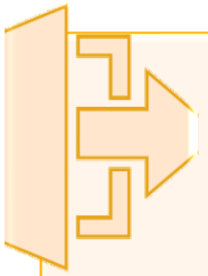
Flexibility to setup required scenario

- 4 distinct flows and their mixes present



High coverage in small duration

- If an instance switches at least once, we consider it to be covered



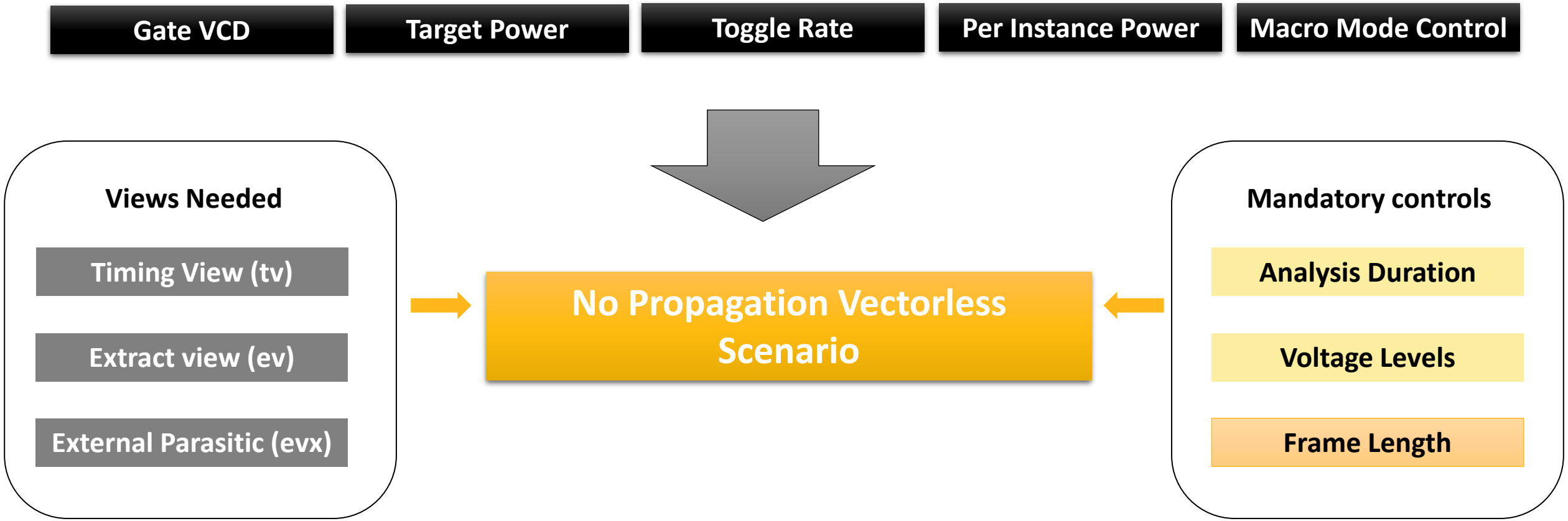
Strictly follows STA input

- Requires STA with high coverage

Tight matching of target power and activity factor

- Match power within ~5%

Overall Flow



Syntax and Examples

scripts/vectorless_dynamic.py

```
scn_npv = db.create_no_prop_scenario_view(timing_view=tv,
extract_view=ev, external_parasitics=evx, **npv_args)
```

Command to launch NPV
Views that get passed to NPV

Pass EV also.
Nets with missing SPEF coverage will have internally calculated net capacitance taken

scripts/args.py

```
tv_args = dict(
    timing_window_files=tw_files,
    logic_graph = False,
    tag="tv",
    options=options)
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=28e-09,
    frame_length=8e-09,
    object_settings=object_settings_npv,
    default_clock = {'policy':'custom', 'period':8e-9},
    tag='npv',
    options=options)
```

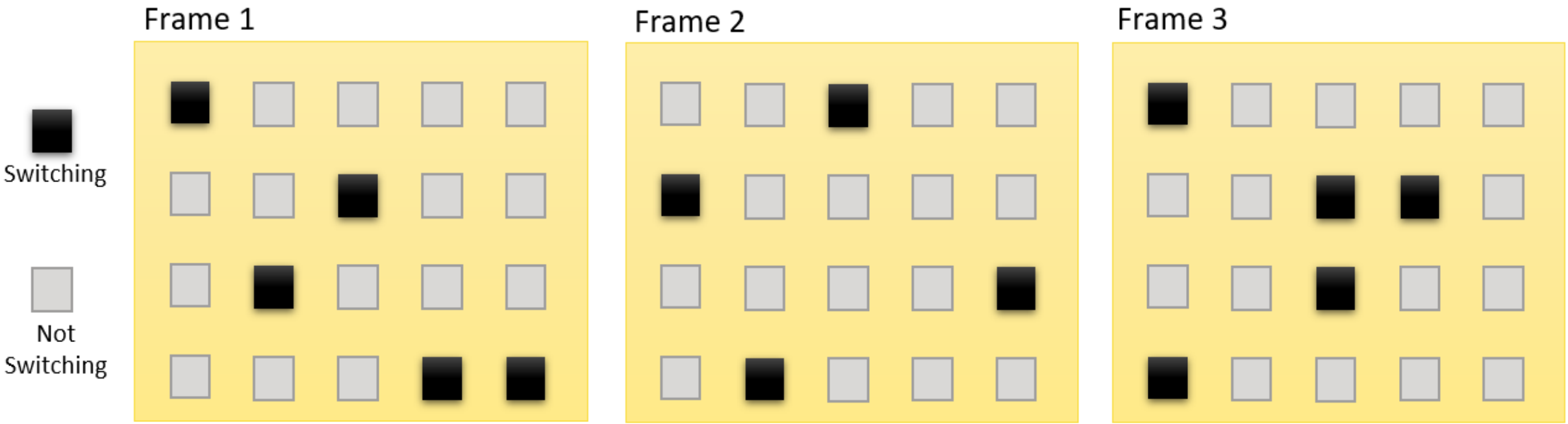
Turn off logic graph **only if the TV is only going to NPV** and no Logic Propagation SCN
Similarly if only NP SWAs are present in static/power flows

analysis_duration is total duration for which scenario is needed
frame_length is duration of each distinct switching sequence

object_settings discussed later
default clock comes into picture when TWF annotation is missing

Concept of Frames

- NPV creates separate switching scenario per frame
- Suggested to use an integral multiple of dominant period in design as frame length
- More number of frames -> more coverage
- Default length of frame is same as analysis duration -> same switching sequence across whole duration.



Concept of Frames

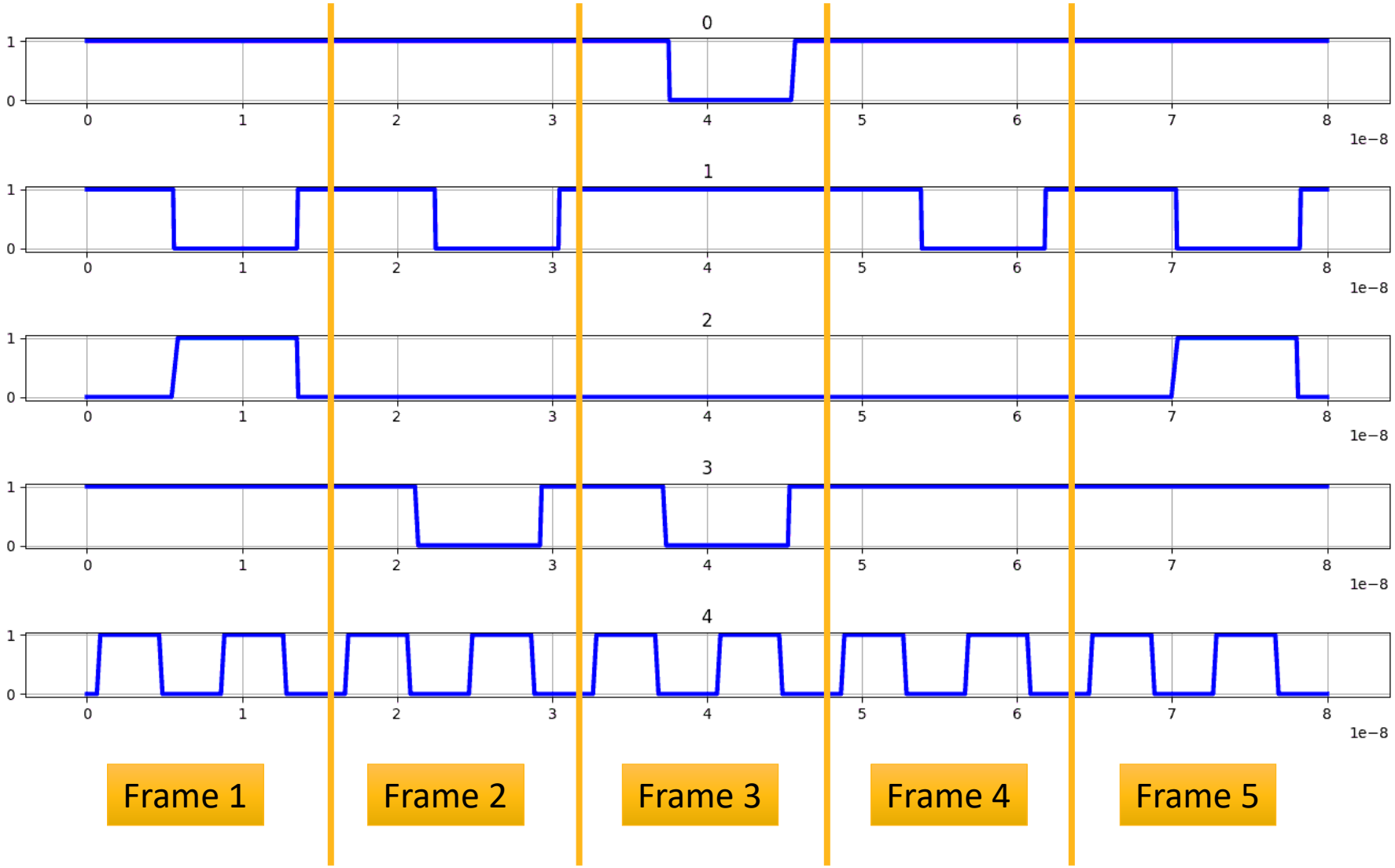
Buffer

OR gate

Register

OAI gate

ICG



Target Power Input

- User inputs target power
  - Design knowledge, input from power team, generic value to start with
- Enough instances selected to switch to meet this target power
  - Area under dynamic current
- Target power met for each frame by switching different sets of instances
- Target power needs to be realistic
  - Not too high, not too low
- Flexible specification of target power
  - Top only, block wise, power domain wise
  - Top/block/domain combinations

Examples : Top and Block

`object_settings` : available in many views across SeaScape
Control different parameters across scopes, i.e. design/top level, block level, master cell, leaf instance level
Here, the parameter is `'target_power'.` Can be given for design and block level

1
```
object_settings = {
    'design_values' :{
        'target_power':{
            Net('VDD') : 2.3}}}
```
2
```
object_settings = {
    'design_values' :{
        'target_power':{
            '*' : 3.3}}}
```

3
```
object_settings ={
    'block_values': [
        {'pattern': 'block1',
            'target_power':{
                '*':0.6}}]}
```
4
```
object_settings ={
    'block_values':[
        {'pattern':'u_ABC/core1*',
            'target_power':{
                Net('VDD1'):0.16}}]}
```

Pattern based matching for block names. Wildcards can be given.
One entry for design_values -> it is a single dict {}
Multiple entries for blocks -> it is a list of dict [{},{}]

```
npv_args = dict(….,object_settings = object_settings)
```

Examples : Domains and Combinations

'*' for domain name means all domains. Can give specific domain
SI units used across SeaScape tools. Here, in Watts

5
```
object_settings = {
    'design_values': {
        'target_power': {
            Net('VDD'): 0.7}}}
```
6
```
object_settings = {
    'design_values': {
        'target_power': {
            Net('vdd1'): 7.2,
            Net('vdd5'): 5}}}
```

7
```
object_settings = {
    'design_values': {
        'target_power': {
            Net('VVDD'): 0.9}},
    'block_values': [
        {'pattern': 'block1',
         'target_power':{
            Net('VDD1'): 2}}}
```

- Power given to both top and block.
- Bottom up precedence is followed
  - An object setting on a more specific scope takes precedence

```
npv_args = dict(….,object_settings = object_settings)
```

One Complete Example

```
object_settings_npv = {
    'design_values' : {
        'target_power' : {
            '*' : 0.3}}
    'mode_control': {
        'mode_probabilities':{
            '_high_energy_mode_' : 0.3,
            '_low_energy_mode_'  : 0.4,
            '_leakage_only_mode_': 0.3}}},
    'block_values' : [
        {'pattern' : 'core3',
        'target_power' : {
            '*' : 0.050}},],}
voltage_levels = {'VDD':1.1, 'VSS':0.0}
```

```
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=40e-09,
    frame_length=8e-09,
    object_settings=object_settings_npv,
    default_clock = {'policy':'custom', 'period':8e-9},
    tag='npv',
    options=options)
npv = db.create_no_prop_scenario_view(timing_view=tv,
external_parasitics=evx, extract_view = ev, **npv_args)
```

'mode_control' is for controlling switching of macros.
Need to be enabled separately for NPV
Controlling Macro Switching is explained in detail later
The above example has top and block level power

Toggle Rate Input

- User inputs activity factor or toggle rate
  - Design knowledge and/or experiments
- Instances selected to switch based on toggle rate
  - 0.3 toggle rate results in roughly 30% of instances switching in each frame
- Finer Controls
  - Per cell-type, per block, per master cell
- Special Flow that ensures maximum coverage
  - Frames with mutually exclusive switching
- Instance toggle rate support
  - Annotate toggle rate from third party source
- Controls to configure Multi Bit Flip Flop (MBFF) switching
  - How many bits or what fraction of bits to switch

Examples : Toggle Rate for Top and Block

`object_settings` itself used
Different parameters to control for each cell type

1

```
object_settings = {
    'design_values': {
        'clock_pin_toggle_rate': 1.5,
        'combinational_pin_toggle_rate': 0.1,
        'sequential_output_pin_toggle_rate': 0.15},
    'block_values' : [
        {'pattern' : 'block3',
        'clock_pin_toggle_rate': 2.0,
        'combinational_pin_toggle_rate': 0.2,
        'sequential_output_pin_toggle_rate': 0.3}]}
```

- Clock pin toggle rate includes all types of clock instances
- Clock instances are identified based on timing window file /STA file input
- Instances with output pin marked as clock in STA file are clock instances

```
npv_args = dict(….,object_settings = object_settings)
```

Examples : Controlling clock pin switching

`'always_active_clocks'` : design level parameter
All clock instances will switch all the time, irrespective of what was given as clock pin toggle rate
All sequential instances will have clock switching across all cycles (by default False)

4
```
npv_args = dict(…..
    always_active_clocks={
        'clock_instances' : True,
        'sequential_instances' : True}
….)
```

5
```
object_settings = {
    'leaf_instance_values': [
        {'instances': Instance('inst2' ) ,
        'toggle_rate' :0.6,
        'always_active_clocks=True}]}
```

Example 5 : clock switching control given to leaf instance
Considering 'inst2' to be flip flop, 60% of clock cycles will have CK->Q switching and 40% will have clock switching
If 'always_active_clocks' is not given, 40% cycles will remain in leakage power

```
npv_args = dict(….,object_settings = object_settings)
```

Examples : Ensure Coverage Flow

Special flow
Toggle rate of 0.25 -> 100% instances can switch across 4 frames
Provided there are no slow frequencies in the design outside the frame_length

6
```
npv_args = dict(…..
    ensure_coverage = True,
….)
```

Picking mutually exclusive sets of instances to switch can potentially create mismatched total demand currents across frames and also potential localized switching effects
Mutually exclusive switching is unlikely in real life scenario
Recommended to be used only for toggle rate based flow

```
scn  = db.create_no_prop_scenario_view (<view_inputs>,**npv_args)
```

Examples : Instance Toggle Rate File

npv_TR_only

```
#inst_name -   - #TR -  -  -   -   -
Inst1    -   -   0.3   -   -   -  -  -
Inst2    -   -   0.2   -   -   -  -  -
```

- NPV does take in PowerView as input and honours power diligently. Discussed later
- Special case of PowerView input
- Only toggle rates present in input PowerView
- Create toggle rate file in exact format given
  - toggle rates in the 4th column,
  - '-' present for all other 7 columns
- NPV tries to meet toggle rate
  - more instances of 0.9 TR will switch,
  - less instances of 0.1 TR will switch
- More frames -> tighter toggle rate matching

```
power_files = ['npv_TR_only']
pwr_itr_args = dict(
    power_files = power_files,
    options = options,
    tag = pwr_itr')
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=32e-09,
    frame_length=8e-09,
    object_settings=object_settings_npv,
    default_clock = {'policy':'custom', 'period':8e-9},
    tag='npv',
    options=options)
pwr_itr = db.create_power_view(dv,**pwr_itr_args)
scn_npv = db.create_no_prop_scenario_view(
timing_view=tv, external_parasitics=evx, extract_view =
ev, power_view = pwr_itr,**npv_args)
```

Examples : MBFF Bit Switching Ratio

`mbff_bit_switching_ratio` controls the fraction of output bits to be switching

```
object_settings = {
    'leaf_instance_values': [
        {'instances': mbff3_list,
        'toggle_rate':0.1,
        'mbff_bits_switching_ratio':1}]}
    'leaf_instance_values': [
        {'instances': [Instance('mbff2'),Instance('mbff3')],
        'toggle_rate':1,
        'mbff_bits_switching_ratio':0.5}]}
    'leaf_instance_values': [
        {'instances': Instance('mbff4'),
        'toggle_rate':0.5,
        'mbff_bits_switching_ratio':0.75,
        'always_active_clocks': True}]}
```

7

Consider that `mbff3_list` has N instances
10% switch at a time
Whenever switching, all bits will switch together,

mbff2, mbff3 switch in all frames
only half of the output bits will switch each time,
1 bit switches for 2 bit mbff,
2 bits switches for 4bit mbff

mbff4 switch in half of frames
When switching, 75% bits will switch -> 6 bits for 8 bit mbff.
clock switching for all frames

```
npv_args = dict(….,object_settings = object_settings)
```

One Complete Example

```
object_settings_npv = {
    'design_values' : {
        'clock_pin_toggle_rate': 1.6,
        'combinational_pin_toggle_rate': 0.15,
        'sequential_output_pin_toggle_rate': 0.2,
        'mode_control': {
            'mode_probabilities':{
                '_high_energy_mode_' : 0.6,
                '_leakage_only_mode_': 0.4}}},
    'block_values' : [
        {'pattern' : 'core3',
        'clock_pin_toggle_rate': 2.0,
        'combinational_pin_toggle_rate': 0.3,
        'sequential_output_pin_toggle_rate': 0.4,},],}
voltage_levels = {'VDD':1.1, 'VSS':0.0}
```

```
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=40e-09,
    frame_length=8e-09,
    object_settings=object_settings_npv,
    default_clock = {'policy':'custom',
'period':8e-9},
    tag='npv',
    options=options)
npv = db.create_no_prop_scenario_view(
timing_view=tv, external_parasitics=evx,
extract_view = ev, **npv_args)
```

'`mode_control`' for controlling switching of macros.
Need to be enabled separately.
The above example has top and block level toggle rates

Annotated Instance Power Input

- PowerView is the input
  - Ways to generate explained in static analysis training module
- NPV does match target power from input PowerView
  - Per category power matching is also possible
- Power across region or power density also be met
  - Power density matches between input power and NPV power
- Auto Switching of Macros
  - Macros switch modes found out based on input power
- MBFFs, Clock only Switching of Flops are taken care
  - Automatically find out switching conditions based on input power

Examples : Instance Power File

Example of Instance Power File (IPF) based PowerView creation shown here
There are other ways of creating this PowerView

high_power_mode.py

```
#inst_name power      pin
Inst1      3.45e-08   VDD
Inst2      6.89e-08   VDD
Inst2      1.30e-09   VDDB
```

Each instance's power is annotated via instance power file (IPF)
IPF is typically generated from third party source.
`macro_settings` explained in coming slides.

1

```
pwr_ipf_args = dict(
    power_files = ['high_power_mode.ipf'],
    tag='pwr_ipf',
    options=options)
pwr_ipf = db.create_power_view(design_view=dv,
**pwr_ipf_args)
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=40e-09,
    frame_length=8e-09,
    default_clock = {'policy':'custom', 'period':8e-9},
    macro_settings={'ipf_target_power_pin_policy': 'all'}
    tag='scn_npv',
    options=options)
scn_npv = db.create_no_prop_scenario_view(
power_view=pwr_ipf, timing_view=tv,
external_parasitics=evx, **npv_args)
```

Examples : Auto Switching of Macros

By default, macros do not switch in NPV flows ; they remain at leakage

```
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=40e-09,
    frame_length=8e-09,
    default_clock = {'policy':'custom', 'period':8e-9},
    macro_settings={'ipf_target_power_pin_policy': 'all'}
    tag='scn_npv',
    options=options)
scn_npv = db.create_no_prop_scenario_view(
power_view=pwr_ipf, timing_view=tv, external_parasitics=evx,**npv_args)
```

`macro_settings` ensures that macros switches
Modes for switching picked up intelligently -> total macro power matches with input PowerView

Examples : Tighter Matching of Cell Category Power

By default, total power from PowerView is met in NPV

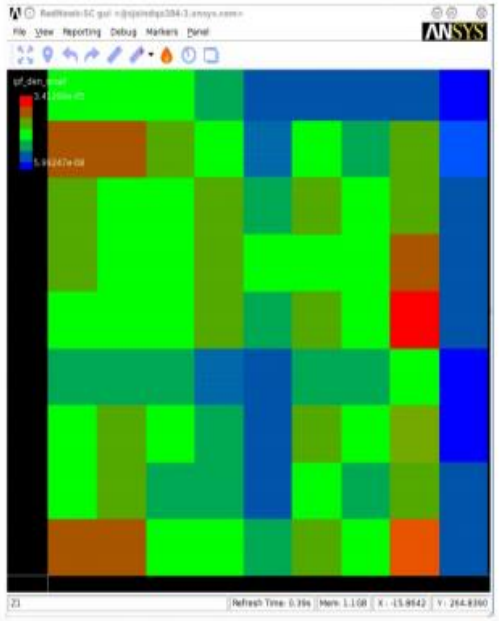
2
```
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=40e-09,
    frame_length=8e-09,
    default_clock = {'policy':'custom', 'period':8e-9},
    macro_settings={'ipf_target_power_pin_policy': 'all'}
    target_power_settings = {'refinement_target' : "Celltype" }
    tag='scn_npv',
    options=options)
scn_npv = db.create_no_prop_scenario_view(
power_view=pwr_ipf, timing_view=tv, external_parasitics=evx,
**npv_args)
```

CellType refinement_target ensures that power across each cell type is met tightly
It takes more effort and hence would take more runtime/memory
The categories considered are clock, combinational, sequential, regbank/mbff, macro

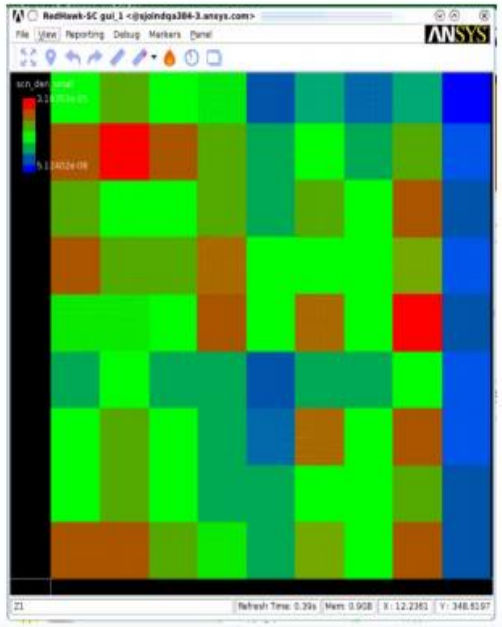
Power Across Region is honoured from Input PowerView

Power Density Comparison

IPF Density Map

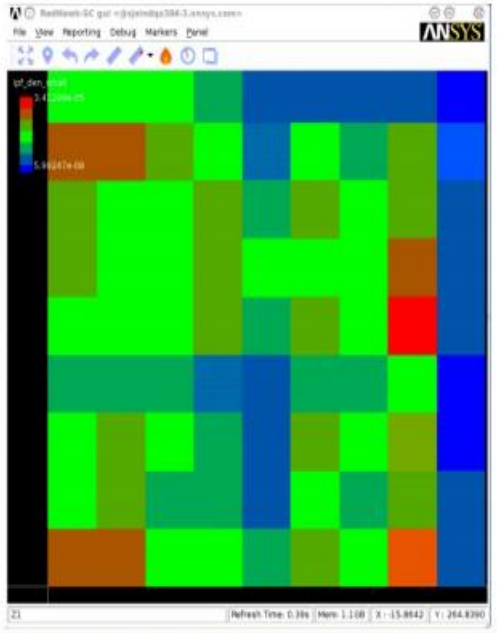


Power Density Map from NPV scenario

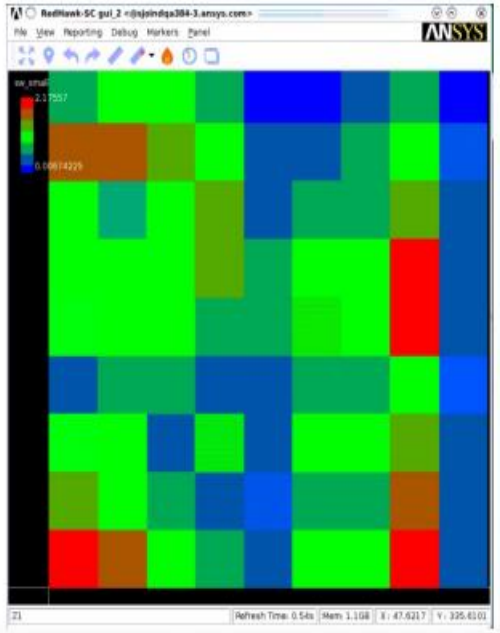


Switching Density Comparison

IPF Density Map



Switching Density Map from NPV scenario



Realize the same switching density as in input power file

Controlling Default Clock Frequencies and Arrival Times

- NPV engine decides on whether an instance switches or not
- When to switch is based on the clock frequency and arrival time, both of which comes directly from timing window file
- When an instance is not present in timing window file
  - Frequency will be taken from default clock
  - Arrival time considered will be across the whole period
- Need to ensure maximum STA file annotation for proper results

Default Clock Handling

Default clock policy values

`'fastest'`

- Fastest among TWF frequencies

`'slowest'`

- Slowest among TWF frequencies

`'dominant'`

- Most used frequency from TWF

`'custom'`

- Specific user input value

- By default, no default clock is present
  - Instances with missing clock information from TWF will not switch
- The argument `'default_clock'` must be set to enable default frequency pickup
- When `'custom'` policy is given, additional key of `'period'` also needs to be provided

1
```
default_clock = {'policy' : 'fastest'}
```

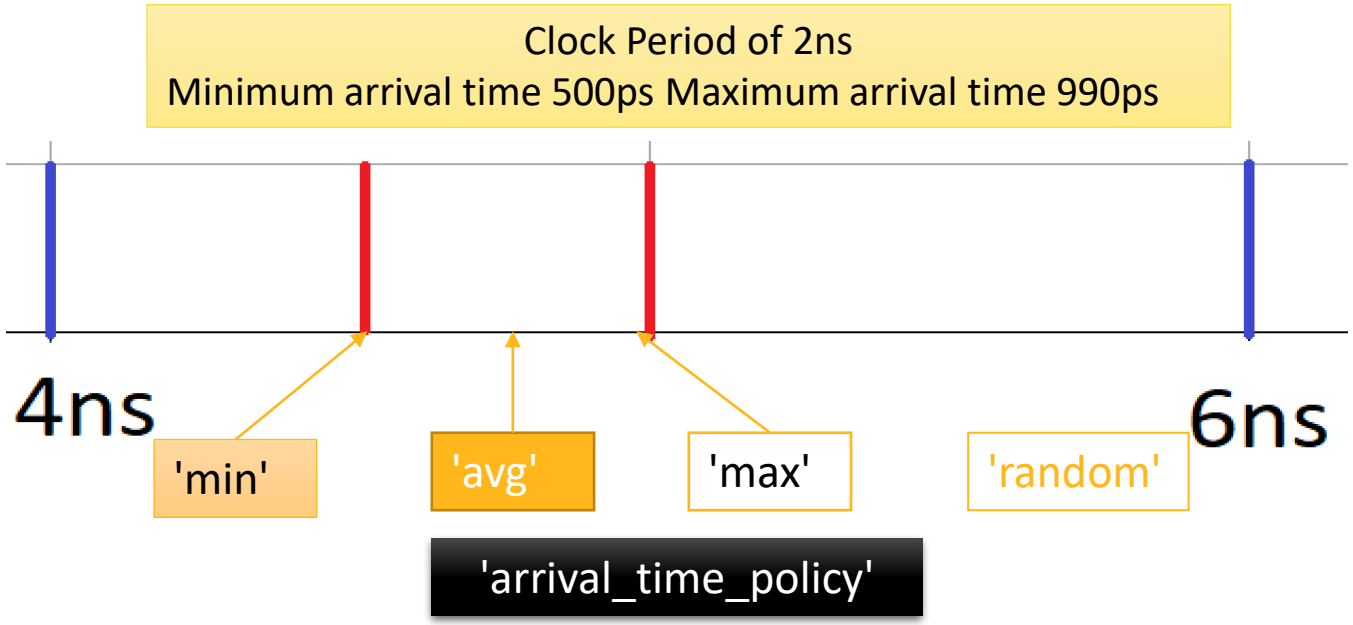
2
```
default_clock = {
        'policy' : 'custom'
        'period' : 1e-09}
```

Arrival Time Policy

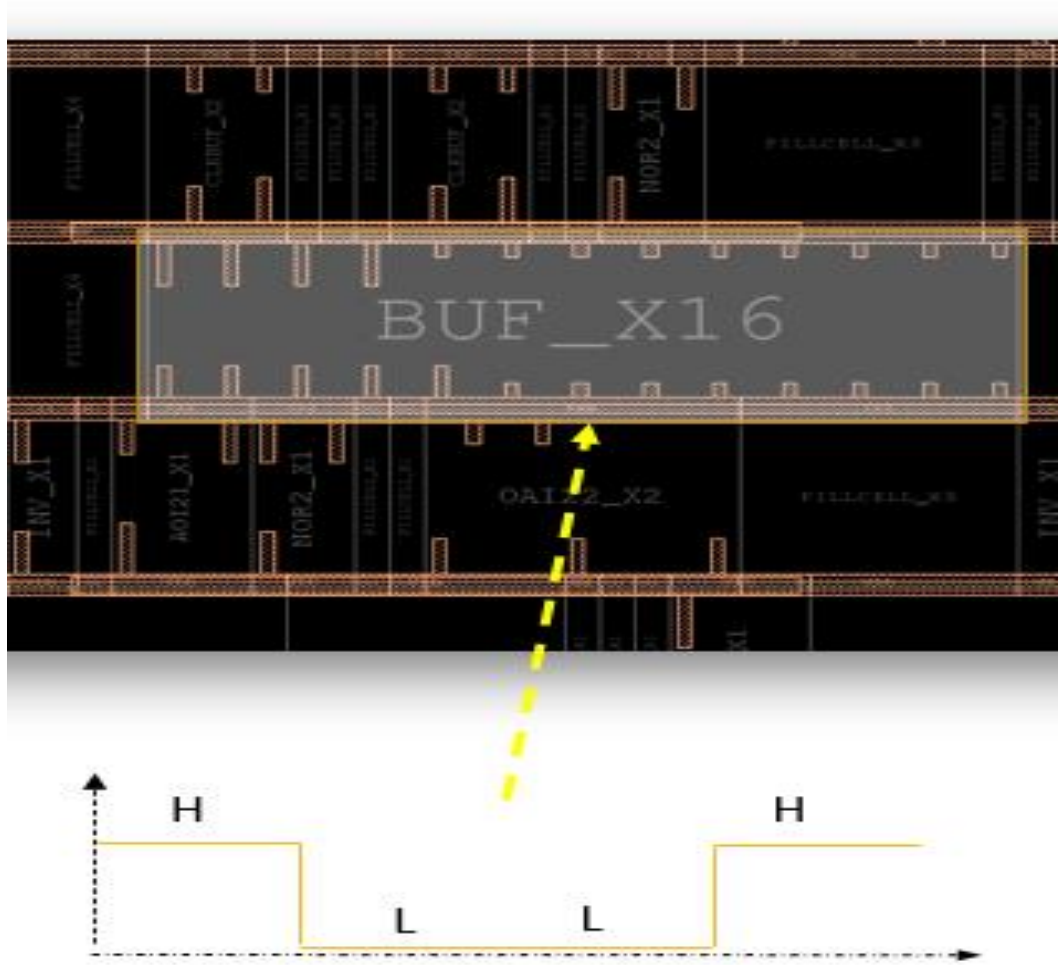
- Events must be placed according to timing window values from timing window file
- Timing window files gives minimum and maximum value for rise and fall event
- NPV to place event within these two values
- User given global switch to decide on where to place
- Separate controls for clock type and data type pins

Default setting

```
arrival_time_policy={
    'clock_path_pins':'avg',
    'data_path_pins' :'random'}
```



Specific Control of Switching Sequence per Instance



- Specific Control of each cycle's switching mode for each master cell/leaf instance
- User can input switching sequence (rise, fall, high, low)
- Per output pin control available for multi output instances
- Active/Inactive mode controls for sequential type instances

Examples : Switching Control

More details on object settings in the section of Controlling Macro Switching Modes

1

```
object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('buf2'),Instance('buf3'),],
        'switching_control' : {
            'switching_sequence' : ['high', 'fall', 'low', 'rise'],},]}}
```

- No switching in first cycle.
- Fall in second cycle
- No switching in third cycle
- Rise in fourth cycle
- Continues the same sequence (-,fall,-,rise,-,fall,-,rise….) for the next cycles until we reach end of analysis duration

```
npv_args = dict(….,object_settings = object_settings)
```

Examples : Mode Control

2

```
object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('inst1'), Instance('inst2')],
            'mode_control' : {
                'mode_sequence' : [
                    '_active_mode_',
                    '_leakage_only_mode_',
                    '_active_mode_)'}, } }
        {'instances' : [Instance('inst1'), Instance('inst2')],
            'mode_control' : {
                'mode_probabilities' : {
                    '_high_energy_mode_' : 0.75,
                    '_low_energy_mode_'  : 0.25}, } }]}
```

An active mode followed by leakage current only and then an active mode
Lots of finer controls possible here. Only some examples mentioned here

High current/energy mode for 75% of cycles. Low for 25%.
Much more info on mode control present in upcoming section

```
npv_args = dict(….,object_settings = object_settings)
```

Examples : Combine Switching Sequence and Mode Control

4

```
object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('inst1')],
        'switching_control' : {
            'output_pin_initial_state' : 'low', },
        'mode_control' : {
            'mode_sequence': ['_active_mode_'], }, }],

        {'instances' : [Instance('inst2')],
        'switching_control' : {
            'output_pin_initial_state' : 'high',},
        'mode_control' : {
            'mode_sequence' : ['_active_mode_', _inactive_mode_'], }, }]}
```

This will happen as (rise, fall, rise, fall, rise, fall…..)

This will be (fall, low, rise, high, fall, low, rise, high …..)

```
npv_args = dict(….,object_settings = object_settings)
```

Examples : Multi Output Pin Instance

3

```
object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('mbff1')],
            'switching_control' : [
                {'pin' : Pin('Q0'),
                'switching_sequence' : ['rise', 'high', 'fall', 'fall'], },
                {'pin' : Pin('Q1'),
                'switching_sequence' : ['fall', 'rise'],
                'switching_sequence_repeats' : False, }]}]}
```

Per pin control of switching enabled
Notice how switching control became a list
Notice logically incoherent sequence here : NPV allows all of these

When `switching_sequence_repeats` is set to False, the last entry will repeat. Here, (fall, rise, rise, rise, rise…..)

```
npv_args = dict(….,object_settings = object_settings)
```

Gate VCD Support into NPV

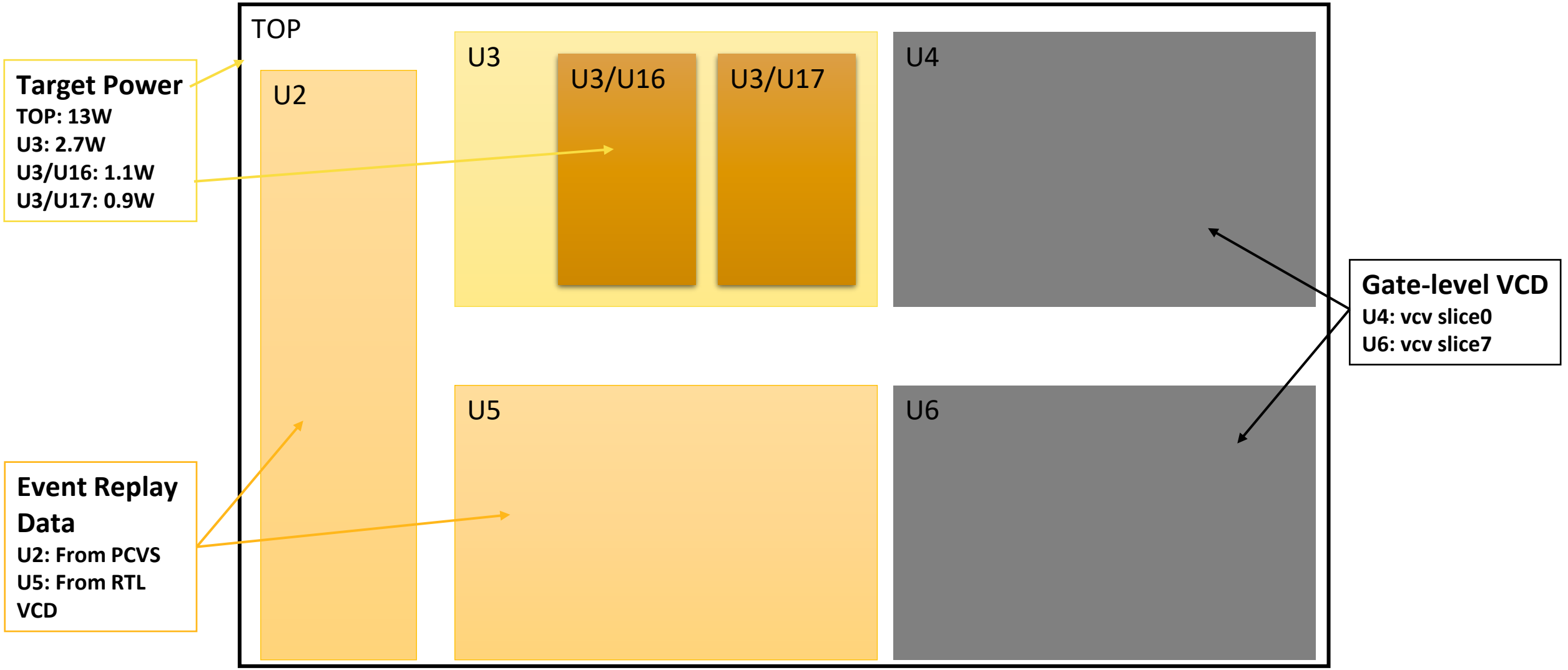
NPV does take in VCD (despite the name No Prop **Vectorless** ☺)

Only Gate VCD. RTL VCD need event propagation which can't be done in NPV

Explained in subsequent training module of VCD Based Dynamic Analysis

Mixed flow of VCD + vectorless also explained there

Hybrid Flow with NPV + Gate VCD



No-Propagation Vectorless(NPV) Scenario

Overview

- Switching of each instance is directly determined without logic propagation
- Events are created per instance, rather than propagated

Benefits

- Very high coverage in short simulation duration
- Highly scalable/parallel - very efficient in memory and runtime
- Actual power is very close to the user-specified target power

Features

- User can specify toggle rates or target power per block
- High-coverage mode to prioritize instances that haven't switched yet

Questions

and

Answers

Logic Propagation
Vectorless Scenario

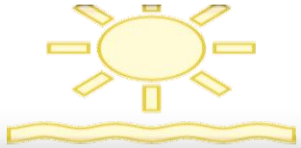
Logic Propagation Vectorless

- Features
- Syntax and Examples
- Clock Propagation
- Data Path Events : Start Points
- Event Time or Arrival Time
- Transition Time
- Per Signal Explicit Control of Events : LSO
- Power Constrained Vectorless Scenario (PCVS)
- Scan Shift Analysis in Vectorless
- Performance and Capacity Advice

Features

Logically Aware Switching Events and Event Times

- Connectivity info utilized and processed



Start Analysis Very Early

- Just DEF and clock root info/SDC needed to setup

In-Built Delay and Transition Time Calculator

- Exact switching time of each event

Presence of Power Constrained Logically Coherent Scenario

- Achieved by controlling ICG switching and hence downstream cone

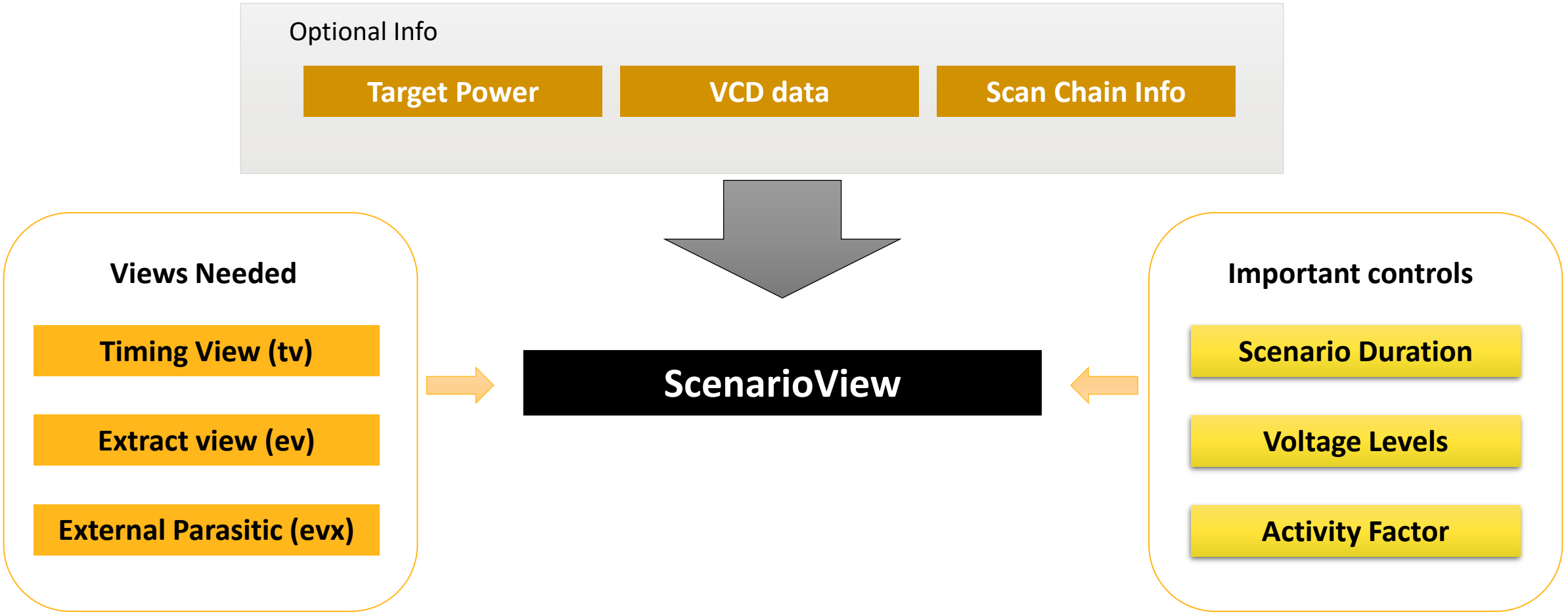
Scan Mode Analysis

- Simulate scan shift mode events without vector

Can become computationally intensive

- Huge DEF blocks, very long scenario duration

Overall Flow



Syntax and Examples

scripts/vectorless_dynamic.py

```
scn = db.create_scenario_view(timing_view=tv,
external_parasitics=evx, extract_view = ev, **scn_args)
```

Command to launch SCN
Views that get passed to SCN

Pass EV also. Nets with missing SPEF coverage will have internal calculated net cap taken

scripts/args.py

```
scn_activity_settings=[{'default_clock_period': 8e-09, 'block_name': '*', 'activity': 0.2}]
scn_args = dict(
    voltage_levels=voltage_levels,
    scenario_duration=48e-9,
    activity_level = scn_activity_settings,
    event_time_precedence = event_time_precedence,
    transition_time_precedence = transition_time_precedence,
    object_settings = object_settings_scn,
    tag='scn',
    options=options
```

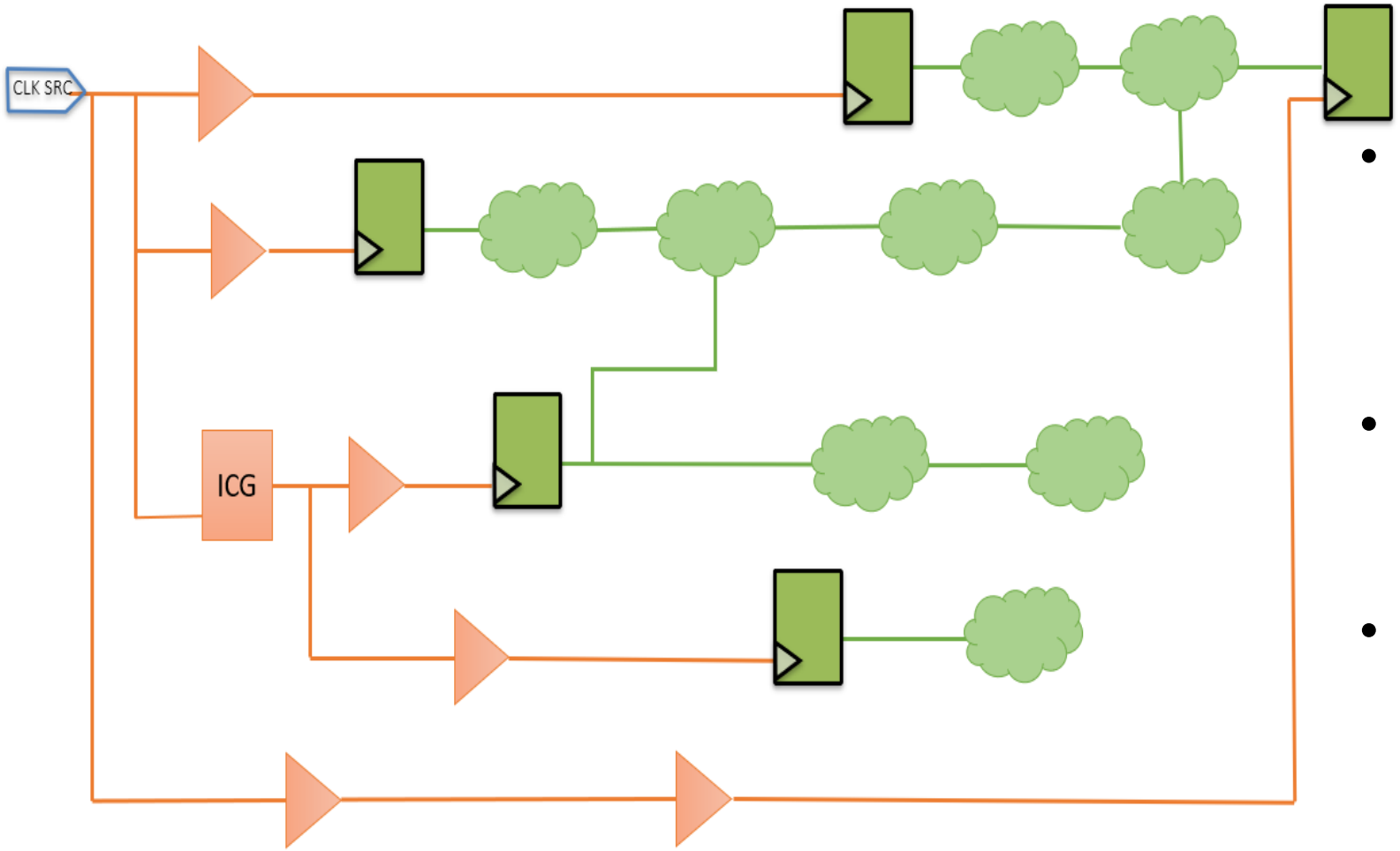
scenario_duration is total duration for which scenario is needed
Activity -> probability of events at sequential output pins

object_settings discussed later
default clock -> when TWF annotation missing/no clocks reached

Clock Propagation

- Clock sources from SDC/STA are considered and then the path traced downstream.
- Events are created at each clock source according to source definitions and propagated downstream
- This is first stage of logic event creation

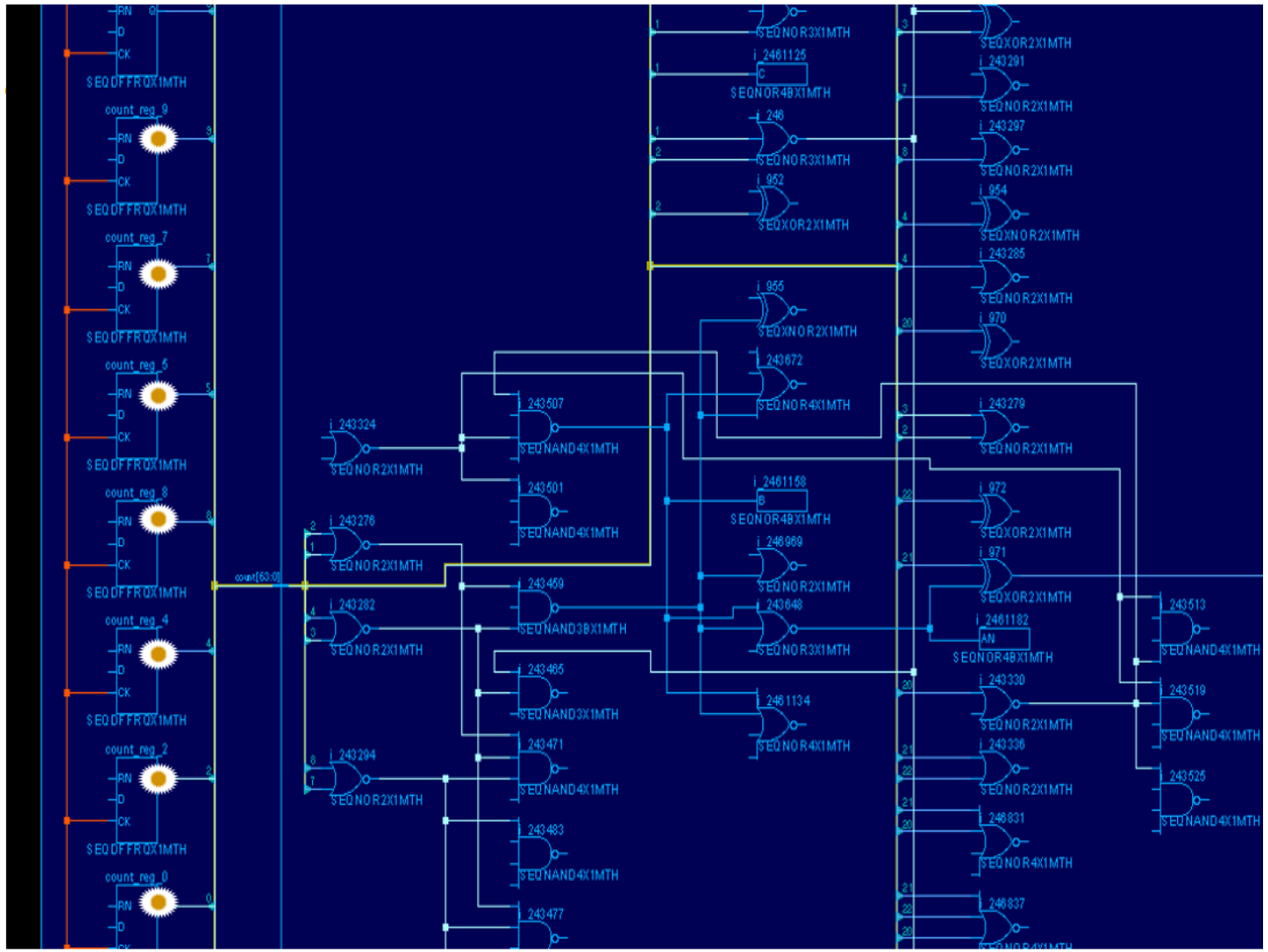
Clock Path Events



- If STA file is present, clock source associations of each instance is checked and corrected (if needed) along the way
- Frequency and duty ratio are considered from the clock source definitions
- When more than one clock comes in at same instance pin, fastest clock gets priority

Data Path Events : Start Points

- Next stage is to do population of events through data network
- Events start from sequential points i.e., register and macro output pins
  - Event Propagation is not done through register and macros.
- In vectorless mode, switching activity of sequential points are determined based on the activity input of user
- Primary Input Points are also start points



Activity Input at Start Points

Global and per block activity for sequential points
`core3` block's sequentials will switch roughly twice compared to rest of design

```
scn_activity_settings=[
    {'default_clock_period': 8e-09, 'block_name': '*', 'activity': 0.2},
    {'default_clock_period': 6e-09, 'block_name': 'core3*', 'activity': 0.4}]
scn_args = dict(…., activity_level = scn_activity_settings)
```

There isn't a direct way to control activity factor per sequential instance
Providing PowerView input is an indirect way -> next slide

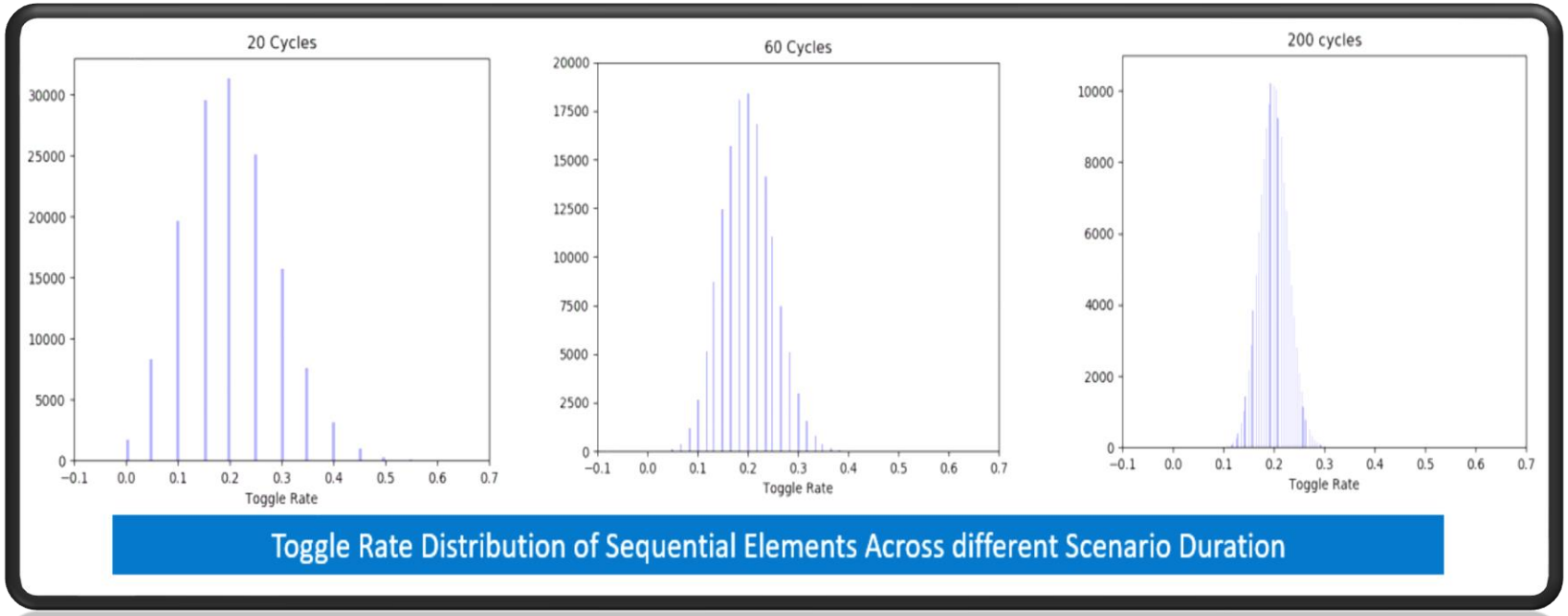
Activity Input at Start Points : PowerView Input

```
scn = db.create_scenario_view(timing_view=tv,
external_parasitics=evx, extract_view = ev, power_view = pwr,
**scn_args)
```

- Static analysis module has details on PowerView creation
- Only activity factor (not power) of sequential points taken from input PowerView
- Register with toggle rate 0.25 in PowerView will have 25% probability of switching
- ICGs, if quiet in PowerView, will remain quiet in ScenarioView
  - The downstream instances will also be quiet
  - If activity of ICG > 0 in PowerView, will be always on in ScenarioView
- All clock instances will remain always on unless controlled by inputs into ScenarioView.
- Rest combinational instances -> don't have any dependence on power or activity from input PowerView

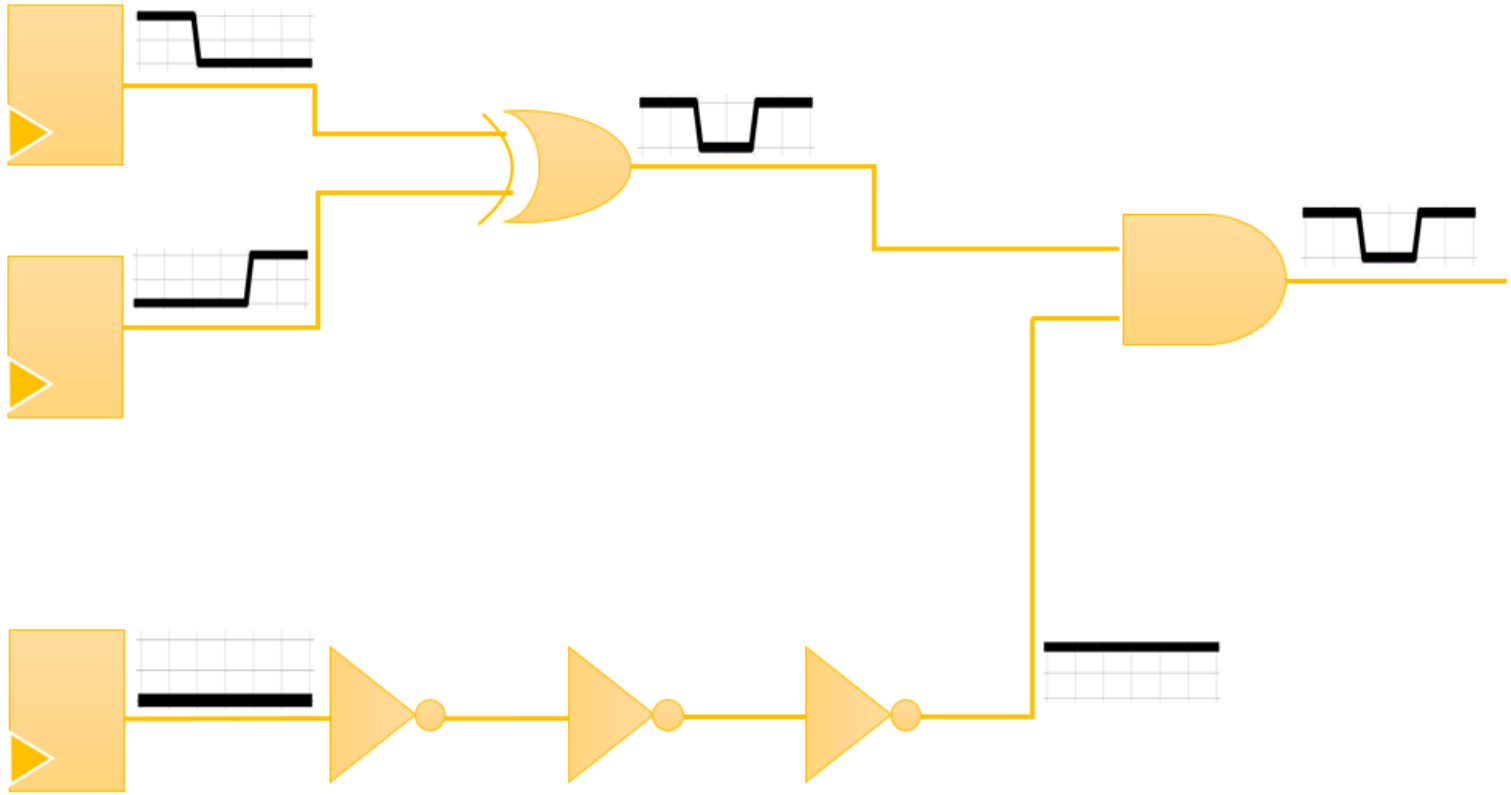
Activity Factor : Probability of Switching

- Activity factor of 0.2
  - 20% probability of switching
  - of each sequential point
  - each output pin of MBFF or macro
  - in each clock cycle
- True randomness
  - akin to coin toss experiment
  - resultant toggle rates of sequentials will be a Guassian function
  - will be bound tighter as number of cycles increases



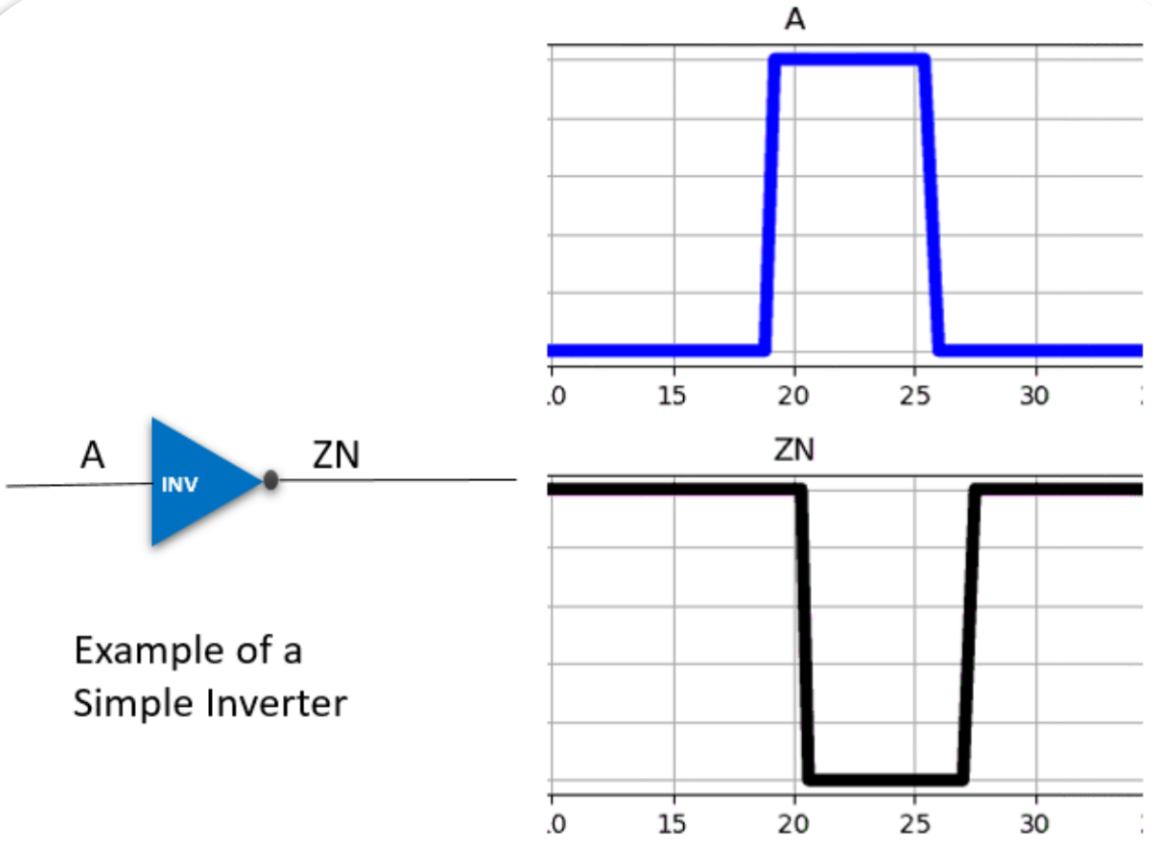
Event Propagation to Combinational cloud

- Events that started at sequential output propagated to downstream cloud
- Like logic simulation, using logic function information from Liberty
- Logically coherent scenario across the design



Event Time or Arrival Time

- Event time at instance output based on input signal time and the calculated delay for the instance
- Delay of each event calculated from liberty file delay tables
- Net delay available with a setting
- User has option to override this calculated delay with STA file arrival time



Example of a Simple Inverter

Settings for Enabling Arrival Time from STA

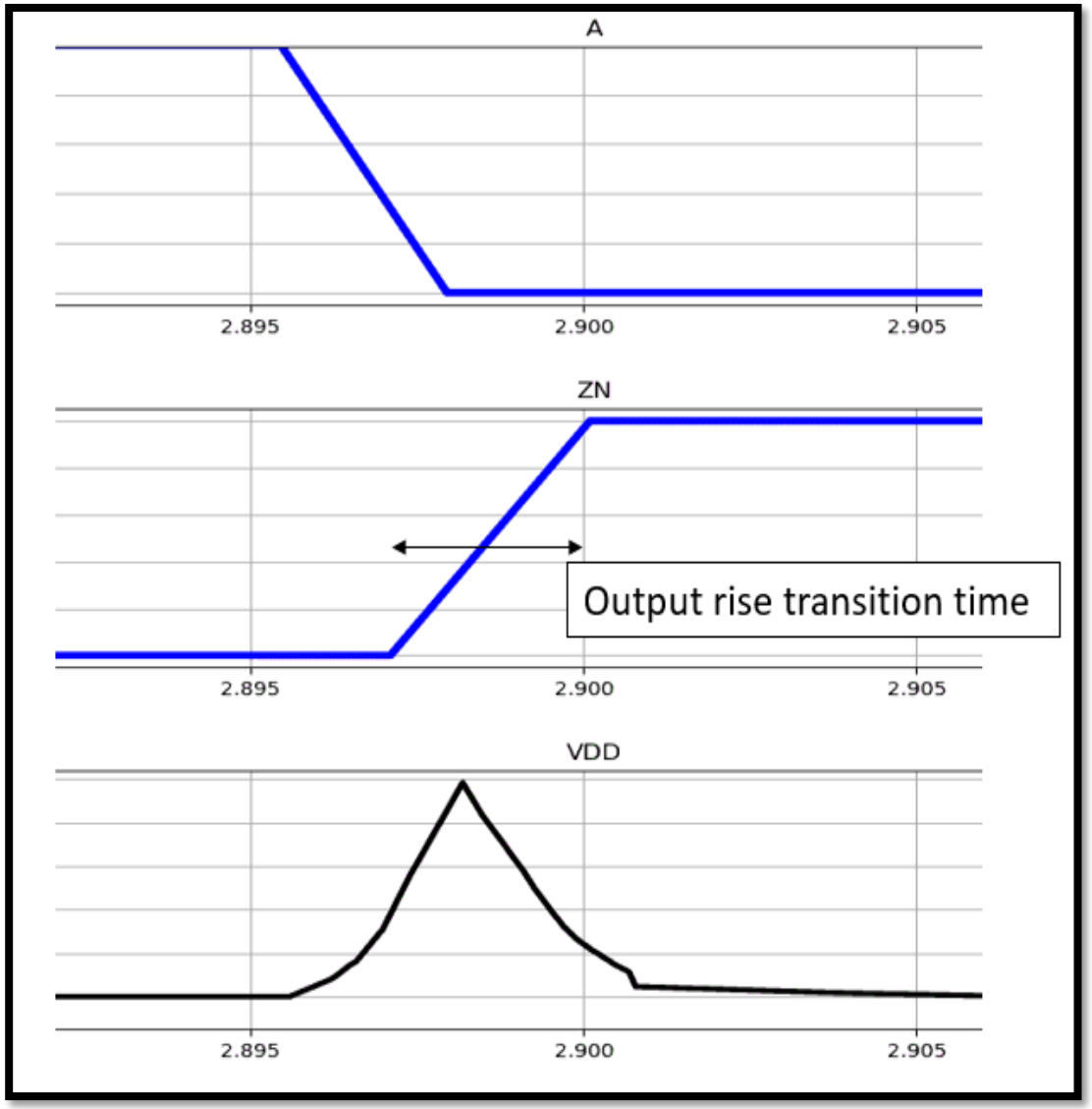
By default, RHSC uses own propagated value, i.e. '`propagated`' for all cell types

```
event_time_precedence = {
    'clock_instance': 'sta_propagated',
    'data_instance': 'sta_propagated',
    'sequential_launch': 'sta_propagated',
    'non_sequential_start_point': 'sta_propagated',}
```

A combination can also be used. Below example shows usage where clock switching times are taken from STA and data switching times from propagated delay

```
event_time_precedence = {
    'clock_instance': 'sta_propagated',
    'data_instance': 'propagated',
    'sequential_launch': 'propagated',
    'non_sequential_start_point': 'sta_propagated',}
```

Transition Time



- Transition time for each event is also calculated directly from liberty file
- User can override this to have STA values taken

Settings for Enabling Transition Time from STA

By default, RHSC uses own propagated value, i.e. `'propagated'` for all cell types

```
transition_time_precedence = {
    'clock_instance': 'sta_propagated',
    'data_instance': 'sta_propagated',
    'sequential_launch': 'sta_propagated',
    'non_sequential_start_point': 'sta_propagated',}
```

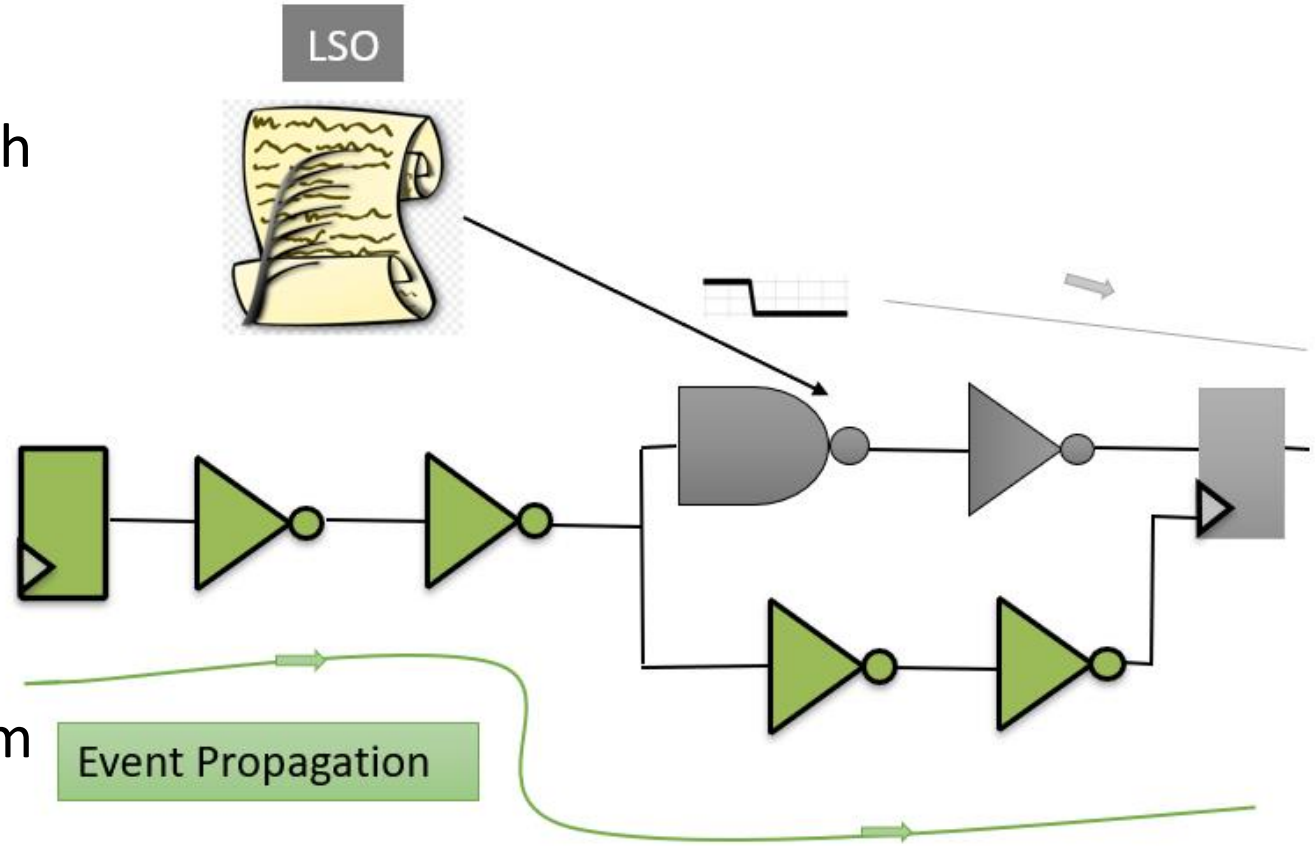
If no transition time data is available in the STA file, a default value is used
Good example is primary inputs

```
scn_args = dict(…..,
    default_input_pin_transition_time = 5e-12,
```

Default value of the above, when not given, is 1ps

Per Signal Explicit Control of Events : LSO

- **L**ogical **S**ignal **O**verride
- Explicit logical signal waveform for each signal pin
- Includes transition time and arrival time
- Overrides events derived from event propagation as well as VCD
- PWL waveform style input
- Event in LSO is propagated downstream



Example : LSO file

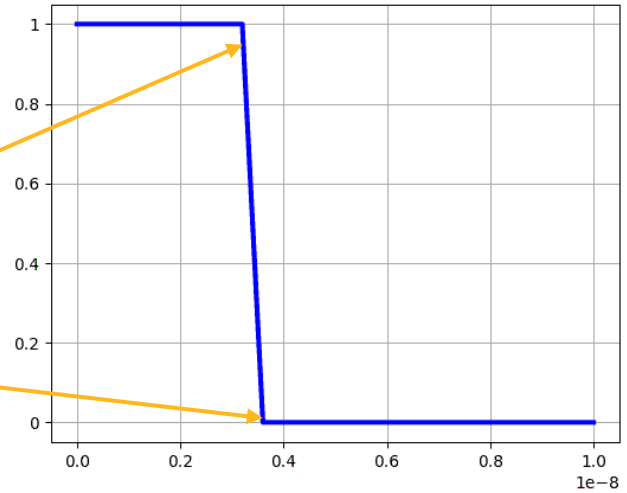
new_override.lso

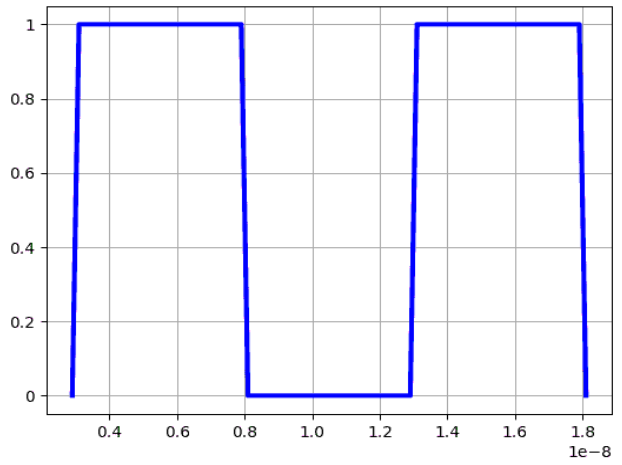
```
$version=1.0
$time_unit=1.0

@ip FF1/Q {
(0.0,1)
(3.2e-09,1)
(3.6e-09,0)
}

@ip FF1/CK {
(0.0,0)
(2.9-09,0)
(3.1e-09,1)
(7.9-09,1)
(8.1e-09,0)
(12.9-09,0)
(13.1e-09,1)
(17.9-09,1)
(18.1e-09,0)
}

@ip ICG3/EN { (0.0,0) }
```





Q: 400ps transition time. Event time is mid point of transition i.e. 3.4ns

CK : Rise at 3ns, fall at 8ns, rise again at 13ns, fall again at 18ns. All of these with 200ps transition time

ICG EN: Actie high enable pin (EN) kept at LOW to have ICG as off

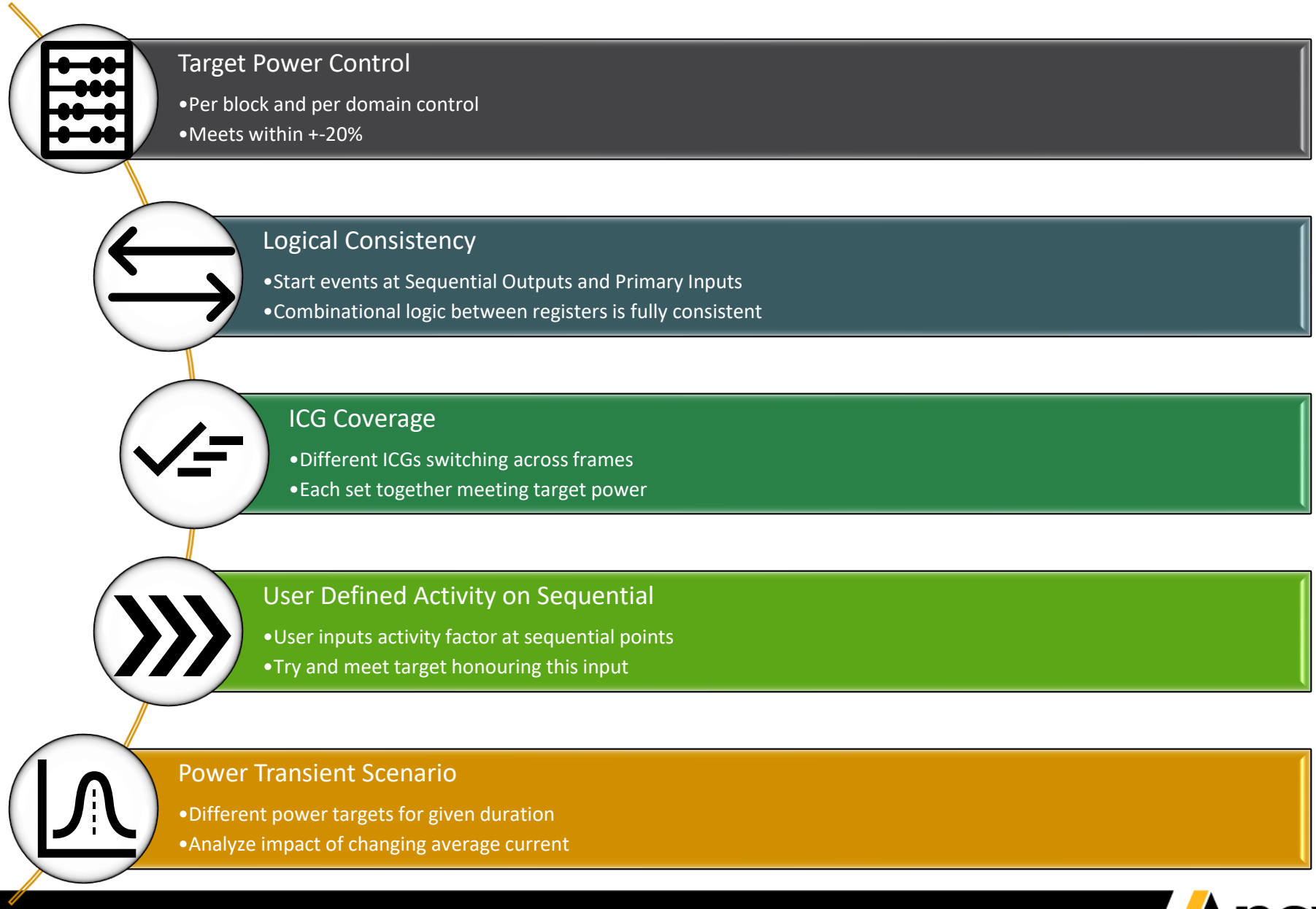
LSO files can be big (millions of instances)
Multiple LSO files and block level LSOs are also possible

```
scn_args = dict(….,lso_files = ['file_name' : './new_override.lso'], …)
```

Power Constrained Vectorless Scenario : PCVS

- Features
- Flow and Core Ideas
- Concept of Frames
- Good Target Power
- Power Transient Scenario
- Macro Control and PCVS
- Settings and Examples

Features

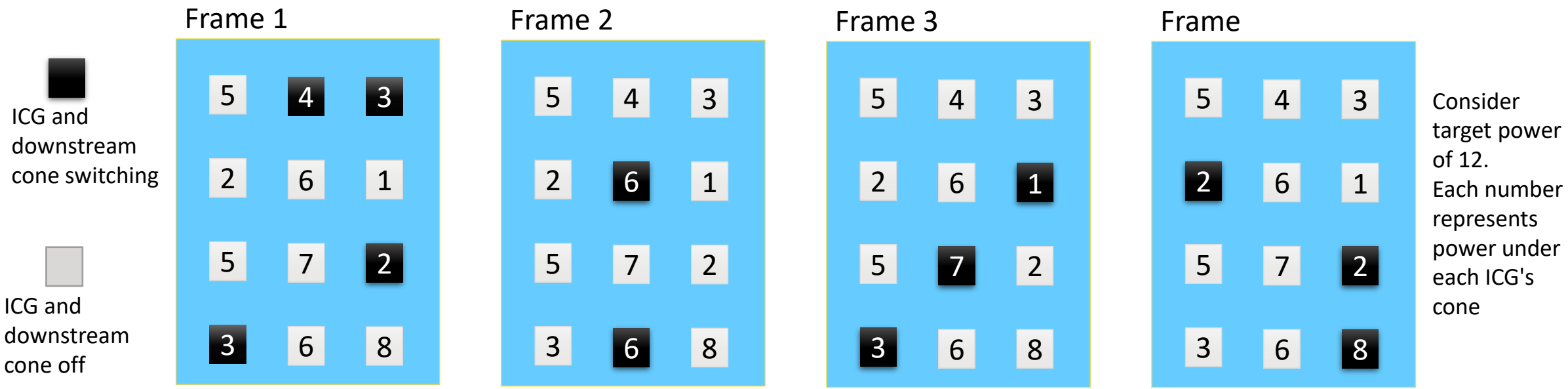


Flow and Core Ideas

- Processing stage done initially
- Downstream cone power of each ICG estimated
- Set of ICGs found out, which are to be gated to achieve target power
- Activity factor also controlled, if needed
- Takes more runtime and memory (~2.5X) compared to a logic propagated scenario without target power constrain

Concept of Frames

- Similar to NPV frame ; different set of ICGs selected for each frame
- Can use multiple clock cycles as one frame, giving chance for more flops to switch
- More number of frames -> more coverage for ICGs
- By default frame length is auto calculated
- Recommended to run PCVS for multiple frames (>10)



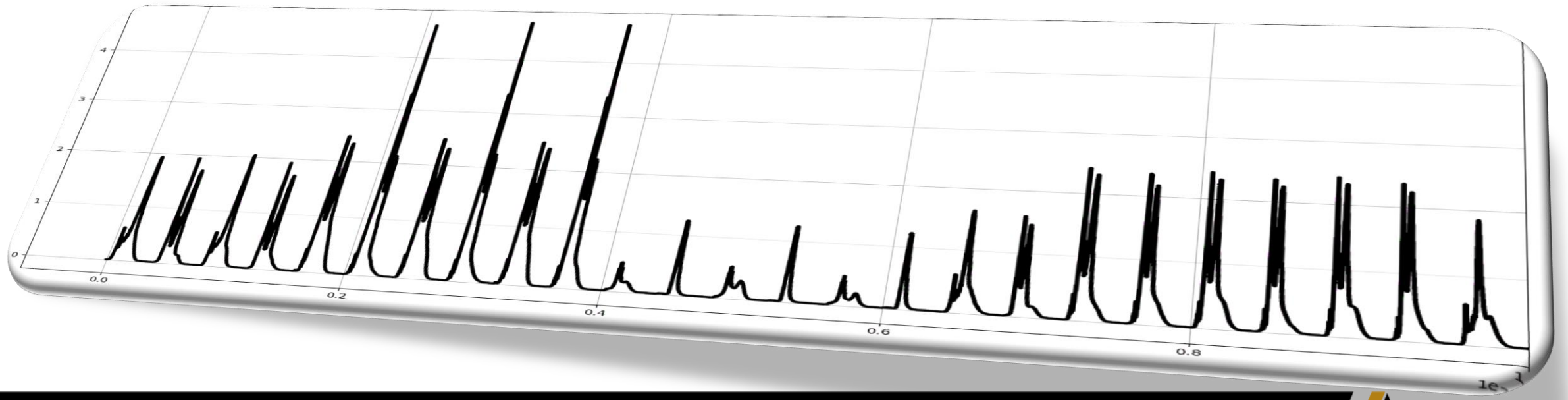
Good Target Power

- A 'good' target power gives best results
- Not too high or too low
- Can be achieved by turning by different set of ICGs
- No extra adjustment of sequential toggle rate required
- An example is explained in the box

- All ICGs and all registers on all the time -> maximum power for Scenario : 100mW
- Leakage power : 5mW
- Clock power, including clock pin : 25mW
- Power from data i.e., flop + combo : 70mW
- Activity factor of 0.2 -> power from data scale down one-fifth : 14mW
- 25+14 = 39mW -> controllable power.
- Target power within this range of 5-44mW (adding leakage power) is achievable by turning off some ICGs.
- Target power of 25mW is good -> achieved by turning off roughly half of ICGs.
- Target power of 50mW -> most ICGs turned on and raise flop activity
- Target power of 10mW -> most ICGs off and lower flop activity

Power Transient Scenario

- PCVS also gives flexibility to create a single scenario which has different average power across different time intervals
- Different target powers and the duration for which they are to be considered is input
- Response to changing current signature can be assessed : high dI/dt can lead to high package drop
- E.g. 5W for first 100ns, then 3W for next 80ns, 7W for next 60ns



Macro Control Settings and PCVS

- Recommended to set explicit modes for macros' switching using object settings
- Macro switching mode control settings explained in next section of this training
- Default macro handling can result in very different powers across scenarios :
  - By default, macro would switch according to the logic state of the various input pins
  - High number of input pins for macros and powers could be very different for various logical combinations
  - Power computed for macro in each of the ScenarioViews could be different
  - This can result in power estimation numbers getting skewed, especially for design with high percentage of macro

Settings and Examples

```
scn_pcvs_args = dict(
    voltage_levels=voltage_levels,
    scenario_duration=48e-9,
    activity_level = scn_activity_settings,
    target_power_settings = pcvs_settings,
    tag='scn_pcvs',
    options=options)
scn_pcvs = db.create_scenario_view(timing_view=tv,
external_parasitics=evx, extract_view = ev,
**scn_pcvs_args)
```

`target_power_settings` mandatory argument.
Everything else remains the same as for a simple vectorless ScenarioView
Presence of `target_power_settings` is what creates a PCV Scenario

Example : Target Power for top/block/domain

Design level single target power

```
pcvs_settings = {
    'power_targets' : {
        Instance('') : {
            '*':{'target_power': 3.5 }}}}
```

Instance('') is the top level or whole design
'*' takes target for all the power domains taken together
Power in is SI units of Watts

Block level and per domain target power

```
pcvs_settings = {
    'power_targets':{
        Instance('u1'):{
            Net('VDD1'): {'target_power': 0.1},
            Net('VDD2'): {'target_power' : 0.3}},
        Instance('u2'):{
            Net('VDD3'):{'target_power':0.05}}}}
```

Target power is assigned bottom up

```
npv_args = dict(….,object_settings = object_settings)
```

Settings : Frame Size

`coverage_interval_duration` is the key for setting frame size explicitly

```
pcvs_settings = {
    'power_targets' : {
        Instance('') : {
            '*':{'target_power': 3.5 }}},
    'icg_coverage_settings' : {
        'coverage_interval_duration' : 16e-09,}}
```

The default threshold is 90%. We can change this

```
pcvs_settings = {
    'power_targets' : {
        Instance('') : {
            '*':{'target_power': 3.5 }}},
    'icg_coverage_settings' : {
        'power_threshold_for_dominant_frequency' : 0.8}}
```

Simple example :
Assume 3 frequencies in design and their power component as
100MHz 5%
500 MHz 35%
900MHz 60%
900MHz most common frequency.
Dominant frequency, as per definition, frequency above which (power_threshold)% of power resides.
Here, it is 500MHz, as adding up powers of frequencies above 500MHz (i.e. 500MHz and 900MHz) gives us 95% power

```
pcvs_args = dict(….,target_power_settings = pcvs_settings)
```

Settings : Further control on Sequential Points' Activity

`toggle_rate_only` enabled for having all ICGs always-on
Achieve target power by controlling only toggle rates

```
pcvs_settings = {
    'power_targets' : {
        Instance('') : {
            '*':{'target_power': 3.5 }}},
    'toggle_rate_only' : True,
    'icg_coverage_settings' : {
        'ensure_coverage' : False,}}
```

`ensure_coverage` is True by default. Different sets of ICGs switches for different frames. Must be False for `toggle_rate_only` flow.

`force_user_toggle_rate` -> activity input by user not modified. Achieve target power by controlling only ICGs

```
pcvs_settings = {
    'power_targets' : {
        Instance('') : {
            '*':{'target_power': 3.5 }}},
    'force_user_toggle_rate' : True,}
```

```
pcvs_args = dict(….,target_power_settings = pcvs_settings)
```

Settings : Power Transient Scenario

Pass a **list** of target powers
Specify `power_interval_duration` for each

```
pcvs_settings = [
    {'power_targets':{
        Instance(''):{
            '*' : {'target_power' : 1}}},
    'power_interval_duration': 32e-9 },
    {'power_targets':{
        Instance(''):{
            '*' : {'target_power' : 0.8},
        Instance('u2'):{
            '*' : {'target_power' : 0.5}}},
    'power_interval_duration': 24e-9 },
    {'power_targets':{
        Instance(''):{
            Net('VDD1') : {'target_power' : 0.6},
            Net('VDD2') : {'target_power' : 0.7}}},
    'power_interval_duration': 40e-9 }]
```

ScenarioView with different average powers across different intervals
A power transient scenario can be created
Total scenario duration : sum of all the `power_interval_durations`

```
pcvs_args = dict(….,target_power_settings = pcvs_settings)
```

Scan Shift Analysis in Vectorless

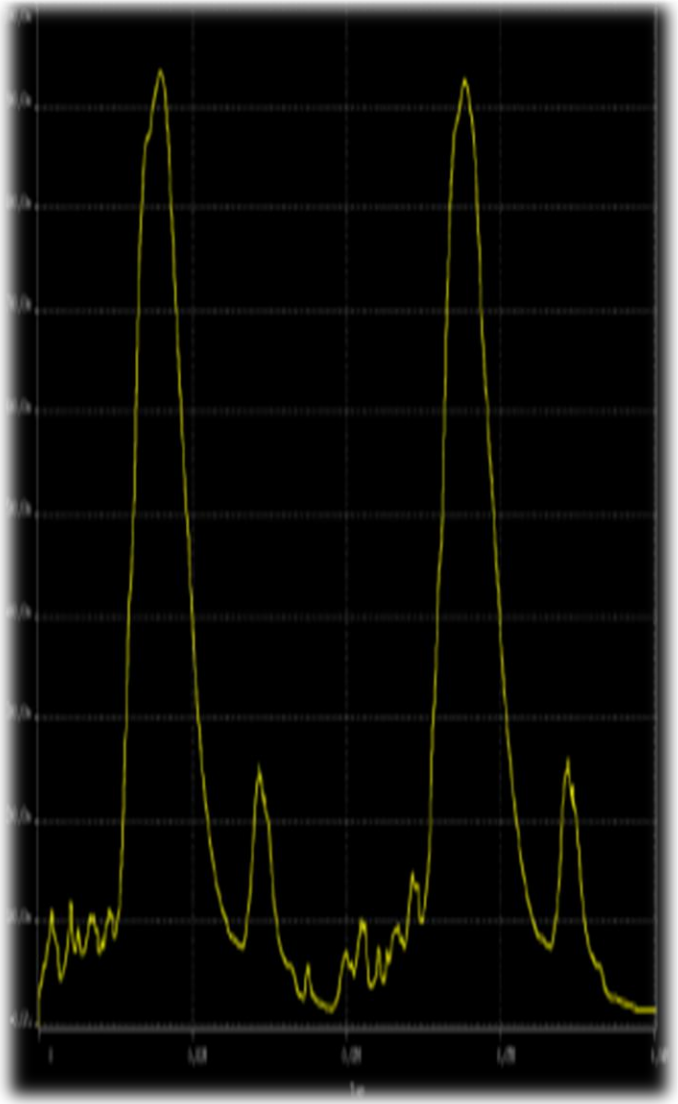
- Importance and Care abouts
- Scan Shift Mode Event Creation
- Scan Chain Data : The main input file
- Important Settings
- Examples
- Handling MBFFs
- Handling Synchronizer Flops

Importance and Care abouts

- Essential to do voltage drop analysis for scan mode even though frequency is low
- Simultaneous switching of lots of registers -> high peak current and change in current
- Coupled with package parasitics -> high voltage drop
- Unintentional switching at combinational paths can also be understood

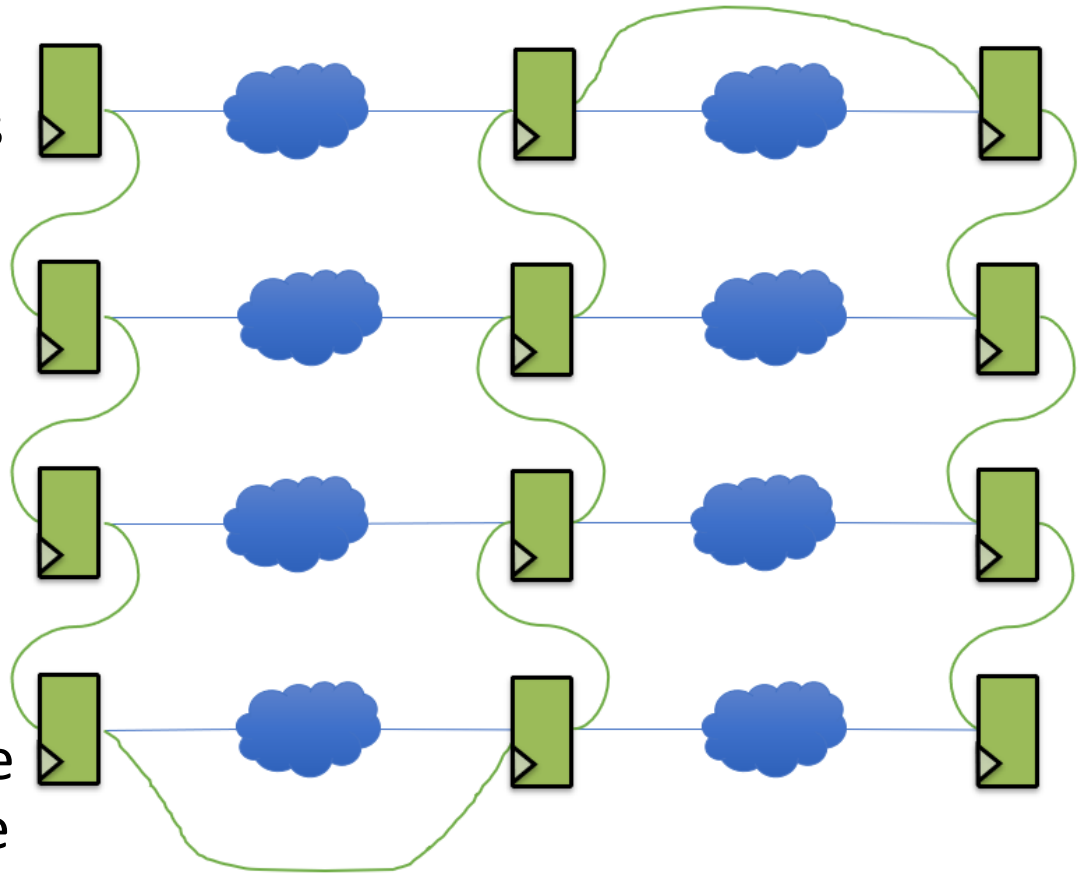
- VCD required to set up flow -> only available very late in design cycle
- RedHawk-SC has the feature of vectorless scan mode analysis
- Do analysis very early and optimize PDN for scan/test mode operation

- Need scan chain data
- Scan clock information as SDC or STA/TWF file



Scan Shift Mode Event Creation

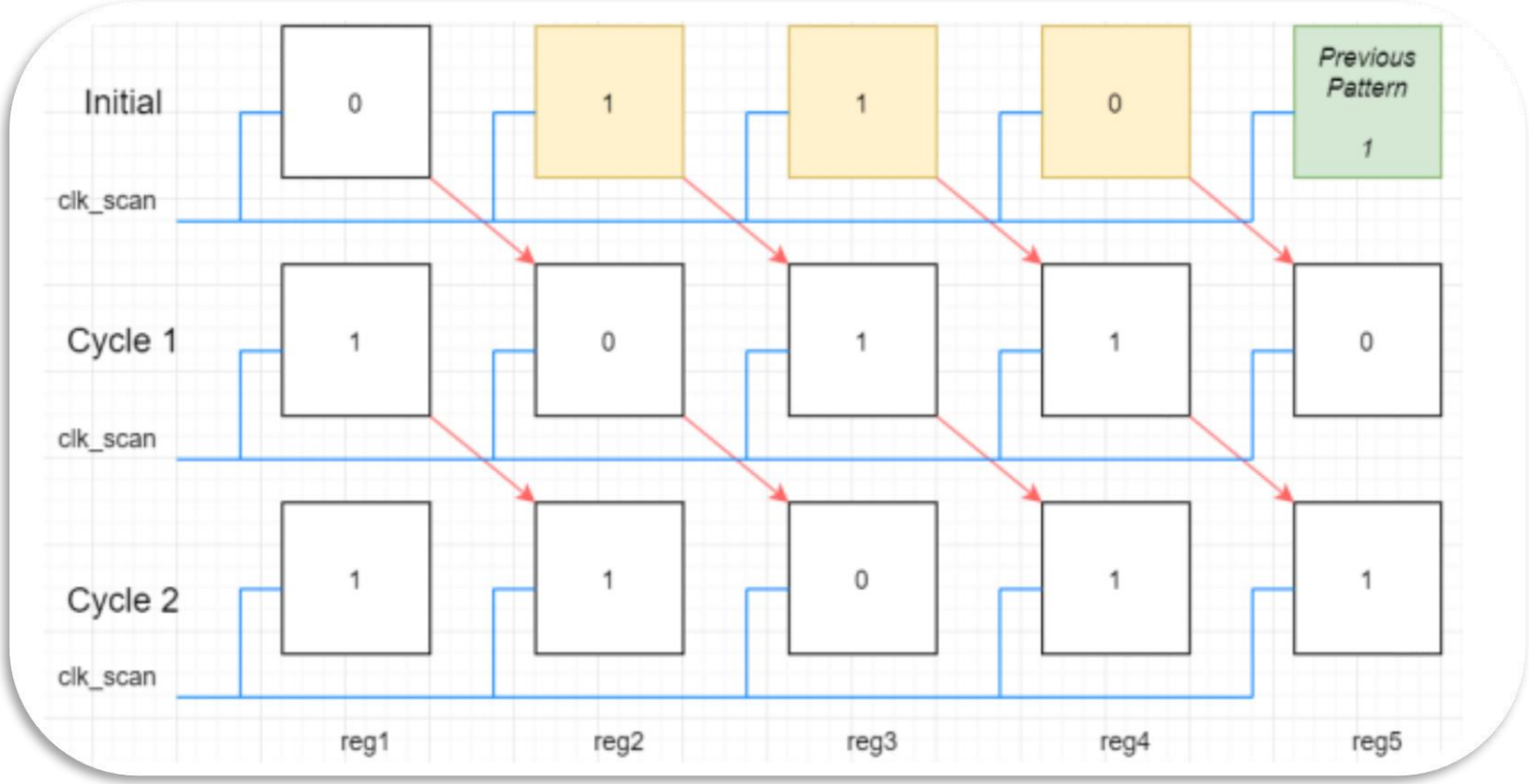
- User inputs scan chain i.e. list of registers in the order of switching
- A bit pattern (e.g. 10010) and number of shifts is also input
- The registers are loaded with the given pattern and then the bits are shifted to corresponding neighboring register for the required number of shifts
- Rise/fall events happen at register output according to bits that gets shifted in
- If scan output is connected to combinational, the corresponding event happening at output will be propagated to the connected combinational logic



Scan Shift Mode Event Creation

Let's look at an example
The scan chain input is
reg1 reg2 reg3 reg4 reg5
Pattern of "110"

Chain length = 5,
Num shifts = 2,
Start Shift = -1
Previous pattern = 1



1. Fill the last -(start_shift) flops with previous_pattern (reg5 ->1)
2. Fill the next pattern_length flops (going towards the head of the chain) with the actual pattern. The LSB of the pattern will be shifted out first (reg4 ->0, reg3->1, reg2->1)
3. Continue to fill the rest of the chain with the pattern, till the first instance is reached. This will give us the initial conditions for all the flops reg1 > 0
4. At each clock triggering edge, shift the data from Flop_N to Flop_N+1. For Flop 0, insert the correct bit such that the pattern is maintained '1' is inserted at reg1 in cycle 1, '1' is inserted at reg1 in cycle 2

Scan Chain Data : The Main Input Data

- The most important data/input for doing a scan mode vectorless analysis
- List of register pins that are part of each scan chain in the right order
- Events are placed/propagated according to this input file
- The data should be available from industry standard tools
- Must be in JSON file format : there are scripts to help with conversion

```
scan_chains = [
    {'file_name': './rhsc_scan_1.json'},
    {'file_name': './rhsc_scan_2.json'}
scn_scan_vless_args = dict(….., scan_chain_data = scan_chains,
num_shifts=4,….)
scn_scan = db.create_scenario_view(timing_view=tv_scan,
external_parasitics=evx, extract_view = ev, **scn_scan_vless_args)
```

`scan_chain_data` : the presence of this argument converts a regular scenario to be scan mode
Ensure to have TimingView created with scan mode inputs

Example : Scan chain JSON file

```
{
    "chain_normal": {
        "instance_pins": [
            "reg1/Q",          # each element is separated by a comma
            "reg2/Q",
            "reg3/Q",
            "reg4/Q",
            "reg5/Q"           # no comma after the last element
    ],
        "pattern": "101",
        "start_shift": 0       # no comma after the last element
},
    "chain_start_shift": {
        "instance_pins": [
            "reg6/Q",
            "reg7/Q",
            "reg8/Q",
            "reg9/Q",
            "reg10/Q"
    ],
        "pattern": "101",
        "start_shift": -1
 }
}
```

- JSON format little strict in terms of syntax and format
- The comments show where to have/not have commas
- If file is not valid JSON format, data not read
- There will be 100s and 1000s of registers for each chain
- Multiple chains can be kept in same file or put in separate files

Important Settings

rhsc_scan_1.json

```
{
"chain_start_shift": {
        "instance_pins": [
            "reg6/Q",
            "reg7/Q",
            "reg8/Q",
            "reg9/Q",
            "reg10/Q"
    ],
        "pattern": "101",
        "previous_pattern": "0",
        "scan_clock": "clk_scan",
        "start_shift": -1
 }
}
```

'pattern' is bit pattern to be propagated through list of pins in scan chain
'10' -> switching happening in all cycles (1010101…. in all registers)

'start_shift' : number of shifts already completed before starting the simulation.
If negative, delay before the first pattern bit is introduced
Typically, negative value indicate number of cycles to wait before starting the first shift-in operation

'previous_pattern' The bit/bits to be used until 'start_shift' is completed, in case 'start_shift' is a negative number

Settings for control of pattern, shift etc.

```
scan_chains = [
    {'file_name': './rhsc_scan_1.json'},
    {'file_name': './rhsc_scan_2.json',
    'previous_pattern': 1,
    'pattern' : '01011',
    'start_shift' : -2 }]
scn_activity_settings=[{'default_clock_period': 8e-09,
'block_name': '*', 'activity': 0}]
scn_scan_vless_args = dict(
    scan_chain_data = scan_chains,)
    num_shifts = 4,
    voltage_levels=voltage_levels,
    scenario_duration=32e-9,
    activity_level = scn_activity_settings,
    tag='scn_scan',
    options=options)
```

The values mentioned here will override that mentioned inside scan chain data JSON files

Recommended to have activity as 0 so only registers that are part of scan chain data switches

`num_shifts` is the total number of shift cycles to create. Take care to adjust scenario_duration accordingly

Only `scan_chain_data`, `num_shifts`, `pattern` and proper timing data are required to get started with scan mode analysis. All the rest are extra settings for explicit control over the events

Handling MBFFs

```
{
"chain_mbff": {
    "instance_pins": [
    "reg_mb1/Q_0_",
    "reg_mb1/Q_1_",
    "reg_norm1/Q",
    "reg_norm2/Q",
    "reg_mb2/Q_3_",
    "reg_mb3/Q_4_"
],
}
}
```

Each MBFF bit should be given separately
Can be taken care easily when creating the input file

Handling Synchronizer Flops

```
{
"chain_synchronizer": {
    "instance_pins": [
        "reg1/Q",
        {"ip":"reg2_sync/Q","extra_cycles":2},
        "reg3/Q",
        "reg4/Q",
        "reg5/Q"
    ],
}
}
```

- Synchronizer flops -> cells made of multiple flops connected in series internally with one output pin
- An N-bit synchronizer flop has 'N' flops connected in series
- N clock cycles needed to shift one bit as compared to a normal flop
- Mark synchronizer flops by specifying `'extra cycles'` in scan chain file
- N-bit synchronizer needs N-1 extra cycles
- At any given moment, we see only the state of the ending flop -> synchronizer flops can have apparent "hidden" bits
- The example shows 3bit synchronizer

VCD Input into Logic Propagation ScenarioView

Logic Propagation ScenarioView can take VCD input

Read in RTL VCDs and propagate the events at sequential points to downstream combo cloud

Mix flows of VCD/Vectorless supported

Will be discussed in the upcoming training module of VCD Based Dynamic Analysis

Logic Propagation Scenario : Performance and Capacity Advice

- All logical events and their delays and transition times are to be propagated across the full design
- Partitioning of design can only be done across DEF hierarchies
  - custom partitioning will make it difficult to maintain logical coherence
  - Huge DEF blocks will result in bottlenecks
- Further, very long scenario duration (say 100 clock cycles) can also result in similar bottlenecks
- ScenarioView for very long duration can result in high runtime and high peak worker memory
- PCVS creates two more ScenarioViews internally : takes 2.5X more time and memory
  - Advised to be used mostly for blocks

Logic Propagation Vectorless Scenario

Overview

- Logically coherent events created by propagating from sequential output points
- Delays and transition times of each event computed and propagated

Benefits

- Very close to actual logical events
- Simulate actual logical paths : analyzing the timing impact of voltage drop

Features

- Power Constrained Vectorless Scenario that is logically consistent
- Scan mode DvD analysis without any sort of VCD input
- Setup early without STA file input

Logic Propagation vs No Propagation ScenarioView

- Close Target Power Match
- Long Scenario Duration
- Big DEF Blocks/Full Chip
- High Coverage in Fewer Cycles
- Annotated Power File Input

No Propagation

- Target Power with Logical Coherence
- Early runs without STA file
- Small DEF blocks
- Delay Aware Propagation
- More Realistic Paths/Events

Logic Propagation

Controlling Macro Switching

Controlling Macro Switching

- Macro Switching Modes : Importance
- Three Sources of Current
- Mode Control : Per Scope Control
- Mode Sequence and Mode Probabilities
- Mode Names, Implicit and Custom Modes
- Controlling When to Apply Modes
- Examples

Macro Switching Modes : Importance

Macro -> Memory, Hard IP, Third Party IP etc.

Single instance that occupies much space and takes high current

Most of the current is internal current -> depends on input pin conditions

Difficult to control this current as state of input pins can vary much

RedHawk-SC allows fine grained control over each macro and it's each clock cycle

Three Sources of Current

APL

- Actual current signatures of macro captured via some Ansys utility, e.g. sim2iprof
- AVM: configuration file with current waveform parameters that user input
- Explicit names present for each mode e.g. READ, CMP etc.
- Per Cycle Currents: inactive edge currents not present separately

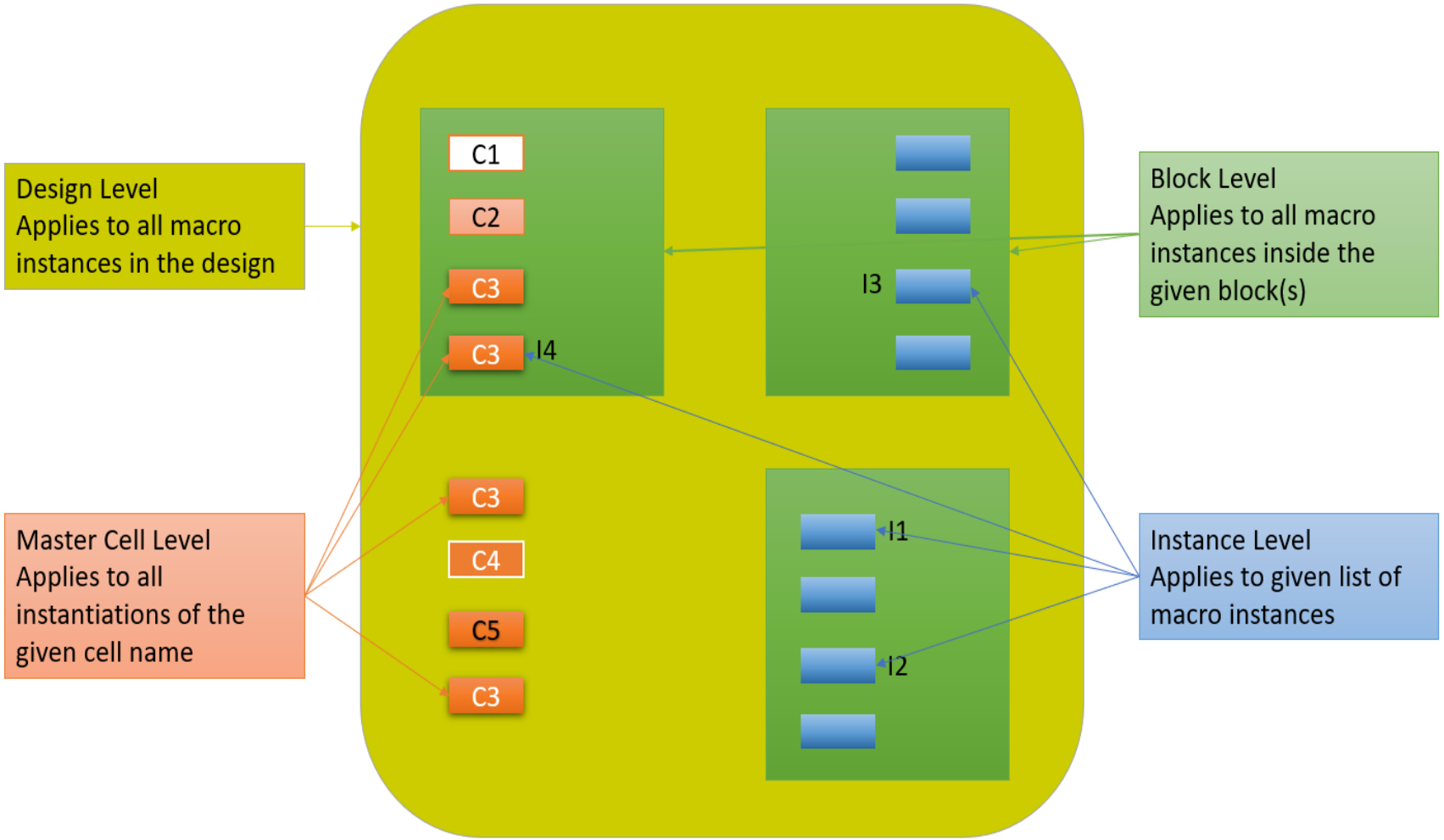
CCSP

- Liberty Composite Current Source (CCS) Power format
- Current waveforms in liberty files for each logic state/transition time/load cap combinations
- Named modes not expected in general, conditional data based on input Boolean functions

NLPM

- Liberty Non Linear Power Model
- Energy values in liberty files for each for each logic state/transition time combinations
- Converted to waveform internally
- Named modes not expected in general, conditional data based on input Boolean functions

Mode Control : Per Scope Control



- Object Settings itself is the input method
- The key `'mode_control'` can be given in any of the scopes.

Mode Sequence and Mode Probabilities

Two broad ways on how the modes can be

Mode Sequence

- Exact Sequence of switching modes
- e.g. READ, STANDBY, CMP, WRITE
- Typically given for cell/instance scopes

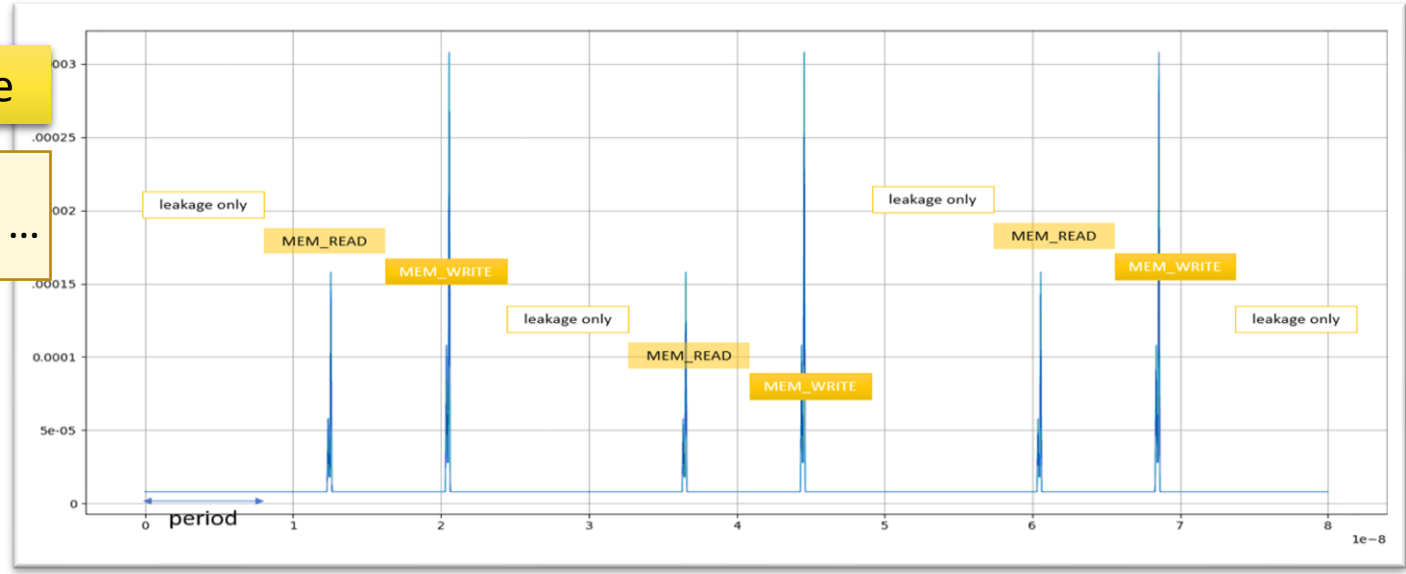
Mode Probabilities

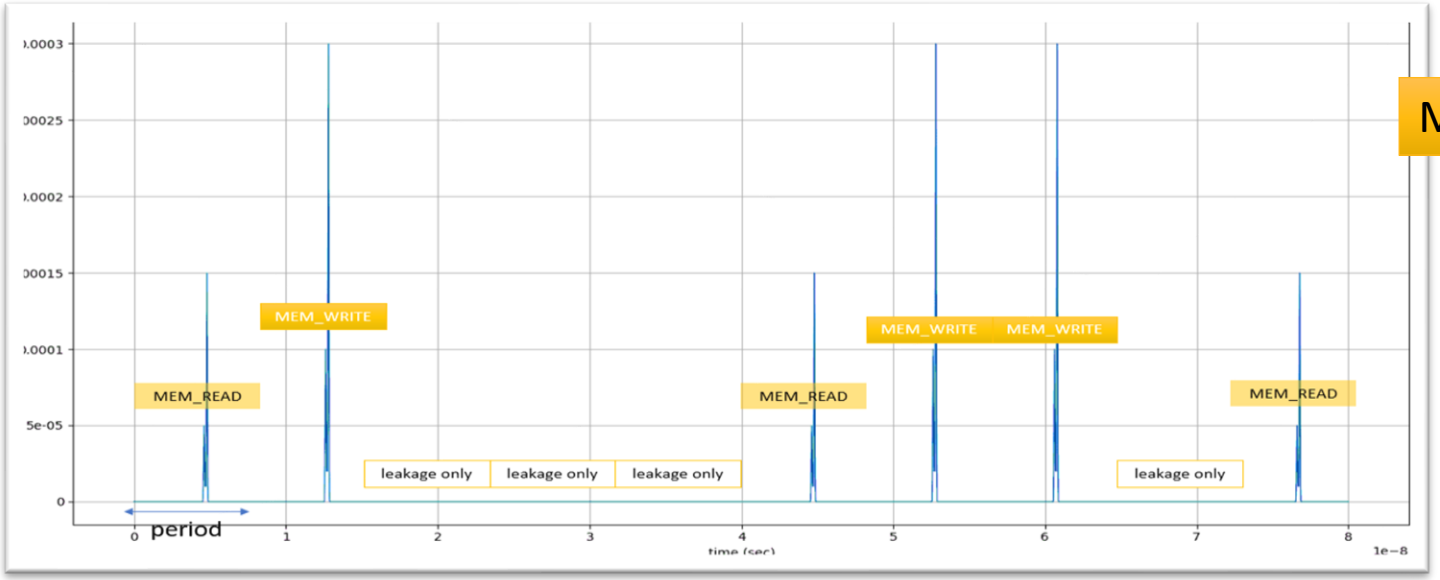
- Probability of occurrence of each switching mode
- e.g. high current for 50%, low current for 50%
- Typically given for block/design scope

Mode Sequence and Mode Probabilities

Mode Sequence

Standby	Read	Write	Standby	Read	Write	...





Mode Probabilities

Mode chosen randomly per active clock edge, based on defined probabilities

Standby	0.4
Read	0.3
Write	0.3

Mode Names, Implicit and Custom Modes

- APL has named modes, 'READ', 'WRITE' etc. -> these are called explicit modes
- RHSC allows a set of implicit names as well. The list being :
  - `'_high_energy_mode_'` : pick highest energy mode
  - `'_median_energy_mode_'`
  - `'_low_energy_mode_'`
  - `'_leakage_only_mode_'` : only a static leakage current for the given clock cycle
  - `'_off_mode_'` : no currents at all : turned off via power switch
- Implicit modes can be given for APL current sources as well.
  - High energy mode will pick the mode with highest energy among the (say, 5) APL modes
- Also define custom modes : mention the needed Boolean expression and associate a name for the created mode

Examples

1
```
object_settings_macro={
    'cell_values' : [{
        'pattern' : 'MEM_CELL_NAME',
        'mode_control' : {
            'mode_sequence' : [
                'ACTIVE_write',
                'ACTIVE_read',
                'standby_ntrig',
                'standby_trig',
                'SLEEP'] }},]}
```

`mode_control` : for setting modes

The same input can be provided to both kind of ScenarioView

```
npv_args = dict(….,object_settings = object_settings_macro)
scn_args = dict(….,object_settings = object_settings_macro)
```

Examples : Implicit Modes

4

```
object_settings_macro={
    'design_values' : {
        'mode_control' : { 'mode_probabilities' : {
                '_high_energy_mode_' : 0.2,
                '_median_energy_mode_' : 0.6,
                '_leakage_only_mode_' : 0.2}, },},
     'leaf_instances_values' : [
         {'instances' : [Instance('ip345')],
         'mode_control' : { 'mode_sequence' : [
             'MEM_write', '_low_energy_mode_', '_median_energy_mode_']}},
         {'instances' : [Instance('mem3'),Instance('mem4'),Instance('mem5')],
         'mode_control' : { 'mode_probabilities' : {
              'MEM_read' : 0.4,
              '_low_energy_mode_' : 0.3,
              '_leakage_only_mode' : 0.3}}},
          {'instances' : [Instance('macro_XYZ')],
          'mode_control' : { 'mode_sequence' : ['_off_mode_']}},],}
```

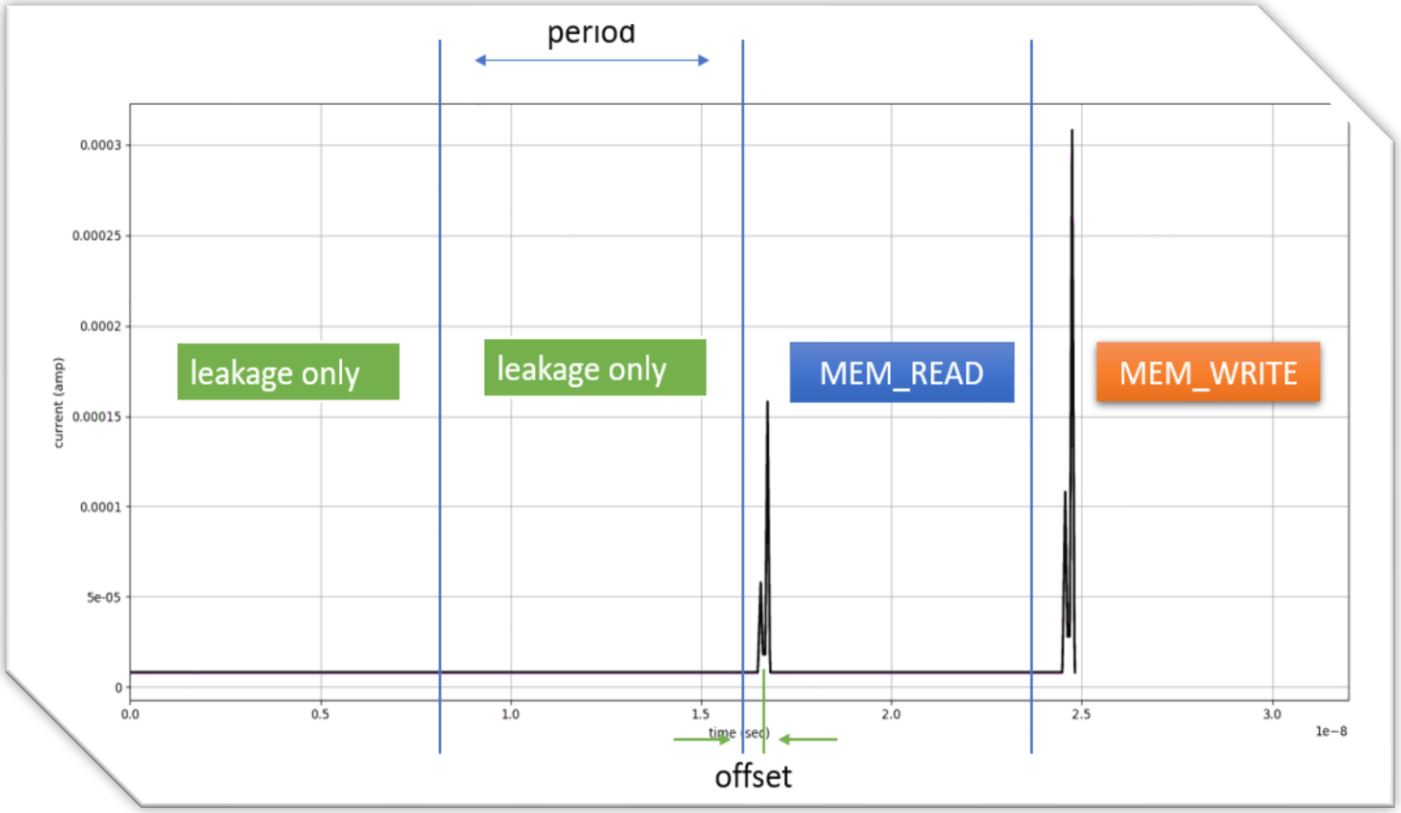
implicit mode can be used for mode sequence and mode probability
Can be mixed with explicit modes

`_off_mode_` cannot be used together with other modes.
Must be the only entry in sequence or set to 1.0 probability

```
npv_args = dict(….,object_settings = object_settings_macro)
scn_args = dict(….,object_settings = object_settings_macro)
```

Controlling When to Apply Modes

- By default, macro switching happens when the active pin, i.e. clock pin switches
- For event propagated scenario, clock switching times can come from event propagation or STA arrival time : there are options in macro mode control to pick between these
- Further, there are controls to explicitly specify period time and offsets for applying currents



Examples : mode sequence

2

```
object_settings_macro={
    'cell_values' : [{
        'pattern' : 'MEM_CELL_NAME',
        'mode_control' : {
            'mode_sequence' : [
                'MEM_write',
                'MEM_read',] }},],
    'leaf_instance_values':[
        {'instances': [Instance('mem1'),Instance('mem2')] ,
            'mode_control' : {
                'mode_sequence' : [
                    'MEM_read',
                    'MEM_write',
                    'standby'],},
                'mode_sequence_repeats' : False},],}
```

`mode_sequence` : for specifying exact sequence

`mode_sequence_repeats` : default True, -> the whole sequence gets repeated
When set to False, the last mode in sequence gets applied till the end.
In the example, `MEM_read, MEM_write, standby, standby, standby…..`

```
npv_args = dict(….,object_settings = object_settings_macro)
scn_args = dict(….,object_settings = object_settings_macro)
```

Examples : mode probabilities

3
```
object_settings_macro={
    'cell_values' : [
        {'pattern' : 'MEM3X*',
        'mode_control' : { 'mode_probabilities' : {
            'MEM_write' : 0.5, 'standby' : 0.5} },},
        {'pattern' : 'MEM2X*',
        'mode_control' : { 'mode_probabilities' : {
            'MEM_write' : 0.75, 'standby' : 0.25} },},],
    'block_values':[
        {'pattern' : 'core2',
        'mode_control' : { 'mode_probabilities' : {
            'MEM_read' : 0.4, 'MEM_write': 0.3, 'standby' : 0.25} },},],
```

`mode_probabilities` is the key for setting probabilities

Just like the activity factor for sequential outputs, mode probability also gets applied as a normal distribution

Recommended to always have mode probabilities add to 1.0
Sum <1, leakage only mode fills the remaining
Sum >1, each mode's value scaled down to have sum as 1

```
npv_args = dict(….,object_settings = object_settings_macro)
scn_args = dict(….,object_settings = object_settings_macro)
```

Examples : Mode Definitions

5

```
new_ram_mode_def = {
    'my_WRITE'   : {'when' : '!we&cs&!byp&!se'},
    'my_READ'    : {'when' : 'we&cs&!byp&!se'},
    'my_STANDBY' : {'when' : '!cs'},}
my_modes_def = [{'name' : 'my_ram_modes', 'modes' : new_ram_mode_def}]

object_settings = {
    'cell_values' :[
        {'pattern' : 'ram*',
        'mode_control' : {'mode_sequence' : [
            'my_READ', 'my_WRITE', 'my_READ', 'my_STANDBY'],
        'mode_definition' : 'my_ram_modes'}}]}
```

The new modes becomes like any other implicit or explicit modes.
Can be used anywhere with mode sequence or mode probabilities

Ensure to pass mode definitions to arguments.

Ensure to use key mode_definitions

```
npv_args = dict(….,mode_definitions = my_modes_def, object_settings = object_settings_macro)
scn_args = dict(….,mode_definitions = my_modes_def, object_settings = object_settings_macro)
```

Examples : Custom Period and Offset

6

```
object_settings_macro={
     'leaf_instances_values' : [
         {'instances' : [Instance('ip345')],
         'mode_control' : { 'mode_sequence' : [
             'MEM_write', '_low_energy_mode_', '_median_energy_mode_']},
         'mode_change' : { 'use_sta' : True },},
         {'instances' : [Instance('mem3'),Instance('mem4'),],
         'mode_control' : { 'mode_sequence' : [
                'MEM_write', '_leakage_only_mode_'],
         'mode_change' : {
             'period' : 16e-09,
             'offsets' : [(3e-09,'CLK', 'r'), (2e-09, 'CLK', 'f')]}
```

STA period used when `use_sta` is set.
Only relevant for event propagated scenario

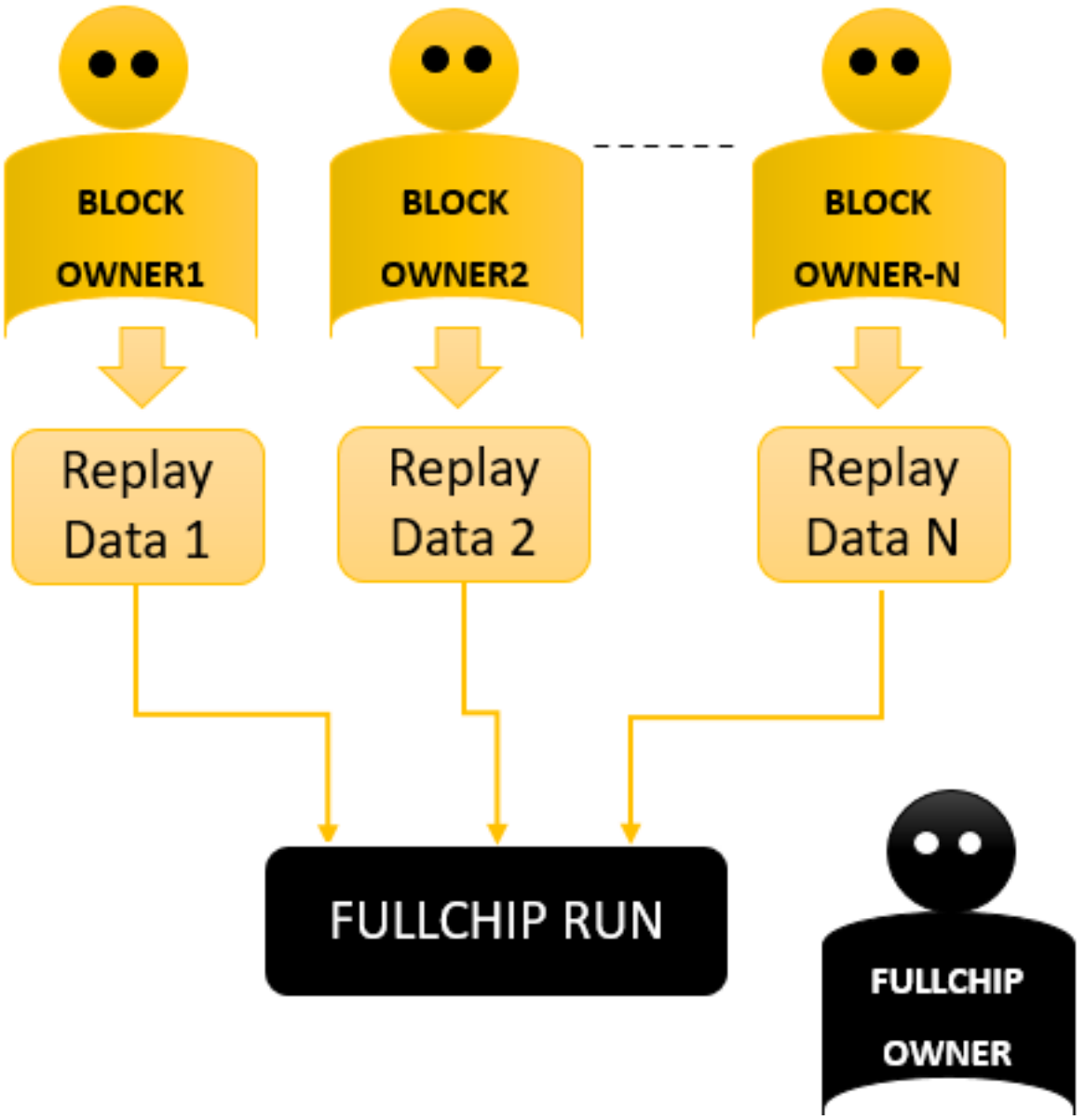
- Clock rise at 3ns and clock fall at 10ns (half-cycle 8ns + given value 2ns)
- For NLPM source, mode applied according to active edge ; the other edge having clock only current
- For APL source, mode applied only at active edge

```
npv_args = dict(….,object_settings = object_settings_macro)
scn_args = dict(….,object_settings = object_settings_macro)
```

Event Replay Flow

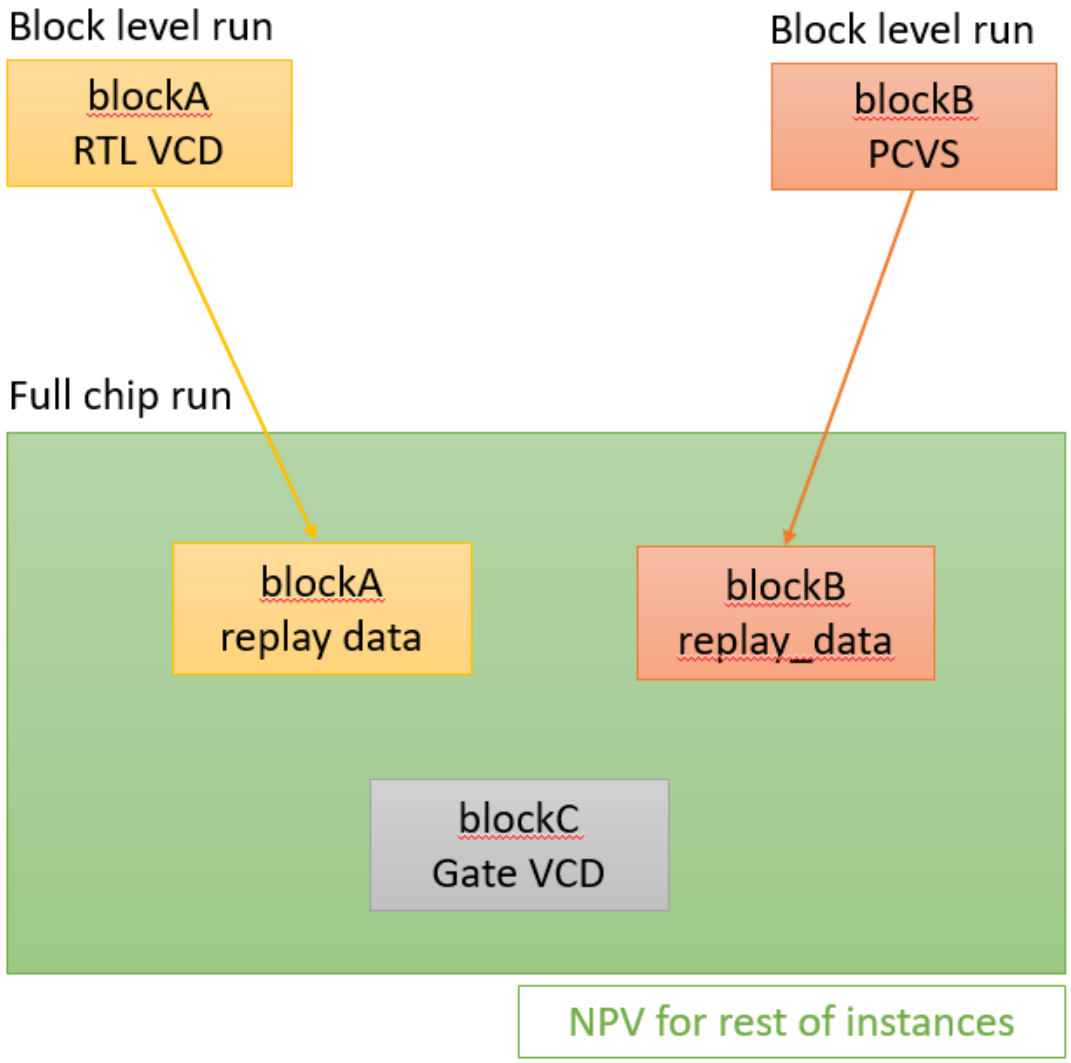
Concept and Need

- Reuse events from one ScenarioView to another
- Integrating/replicating block level setup/events to a top-level run
- Block owner develops Scenarios that are representative/good for DVD -> use these exactly when the block instantiated at higher level
- Easily used across ScenarioView types (Logic Propagation/NPV/PCVS etc.)



Methodology

- Dump event replay data (a directory) from one ScenarioView
- Read it into the next ScenarioView
- All combinations : block -> top, top -> block, top -> top, block -> block,
- All combinations : No Prop-SCN->Logic Prop-SCN, PCVS->NPV, VCD->NPV ….



Examples : Writing out event replay data

API to write out event replay data. Used with any type of ScenarioView

1

```
scn_npv_30tr.write_event_replay_files(
    file_dir_location = 'npv_40ns_30%act_fullchip_replay')
scn_vcd_rtl.write_event_replay_files(
    file_dir_location = 'blockC_replay_data_rtl_vcd_modem_63ns',)
```

Default is top level. Just input directory location for whole data to be written out

2

```
scn_npv_300mW.write_event_replay_files(
    file_dir_location = 'npv_36ns_300mW_core3_replay',
    instance = Instance('core3'))
scn_pcvs_400mW.write_event_replay_files(
    file_dir_location = 'block1_replay_data_pcvs_400mW_89ns',
    instance = Instance('hier3/block1'))
```

Above example for writing out replay for block level
Good practice to have file name with all details

Examples : Reading in event replay data

The argument `event_replay_data` takes in the directory written out

3
```
event_replay_core3 = [
    {'file_dir':' 'core3_replay_data_pcvs_400mW_40ns',
    'instances': ['core3']}]
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=40e-09,
    frame_length=8e-09,
    event_replay_data = event_replay_core3,
    object_settings=object_settings_npv,
    default_clock = {'policy':'custom', 'period':8e-9},
    tag='npv',
    options=options)
scn_npv = db.create_scenario_view(timing_view=tv,
external_parasitics=evx, extract_view = ev, **npv_args)
```

Example for NPV taking in event replay for a block from PCVS
Event replay data takes higher priority over other settings i.e. VCD, toggle rate, target power etc.

Examples : Reading in event replay data

4

```
event_replay_core3 = [
    {'file_dir':' 'ABC_replay_data_npv_IPF_M_32ns',
    'instances': ['block1_ABC','block2_ABC']}]
scn_args = dict(
    voltage_levels=voltage_levels,
    scenario_duration=32e-9,
    activity_level = scn_activity_settings,
    event_time_precedence = event_time_precedence,
    transition_time_precedence =  transition_time_precedence,
    object_settings = object_settings_scn,
    event_replay_data = event_replay_core3,
    tag='scn',
    options=options)
scn = db.create_scenario_view(timing_view=tv,
external_parasitics=evx, extract_view = ev, **scn_args)
```

Default value for `instances` is top level instance.
Can be setup in that way as well i.e. event replay input for whole design

Example for Logic Prop SCN taking in event replay for two blocks from NPV
Event replay data takes higher priority over other settings i.e. VCD, toggle rate, target power etc.

Questions

and

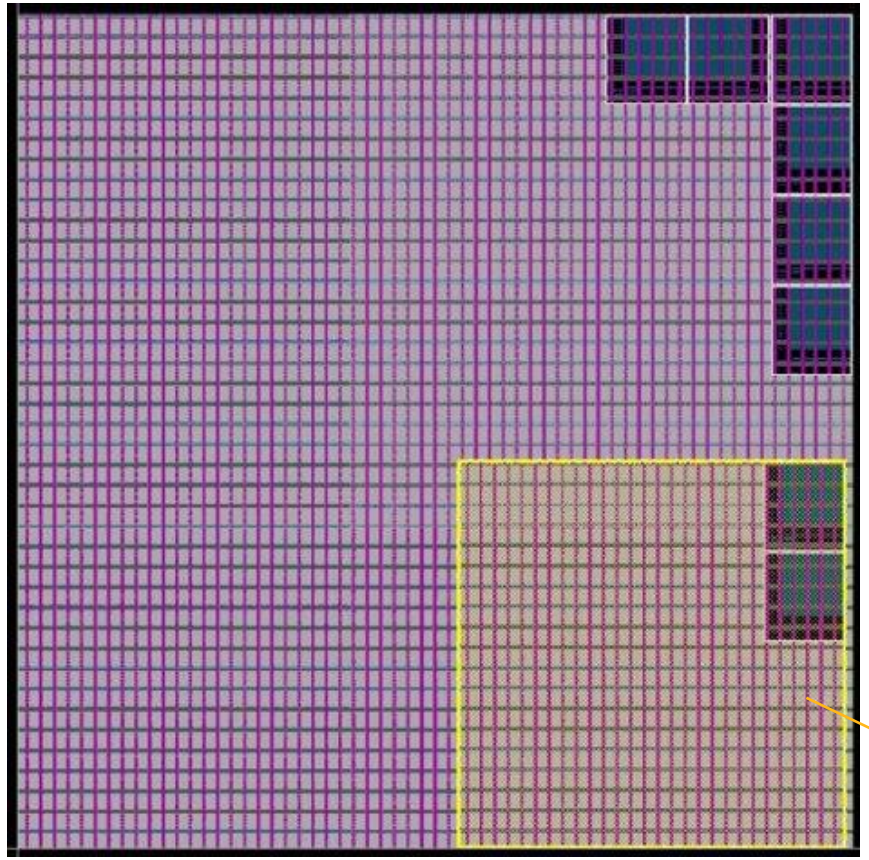
Answers

Training Lab

Training Labs

- Training Testcase Overview
- Getting Familiar with Training Lab Scripts
- Vectorless Analysis Settings : NPV and PCVS
- Log file review
- Switching Summary Report
- Dynamic Voltage Drop Reporting
- GUI
- Interactive Shell

Training Testcase Overview



Galaxy Testcase Details	
Instance count	1.2M
Node count	5M
Domains	1 Always-ON VDD domain 1 Power-gated VDD domain 1 GND domain
#Memories	8
Tech Node	45nm
#Layers	10

Power Gated Block

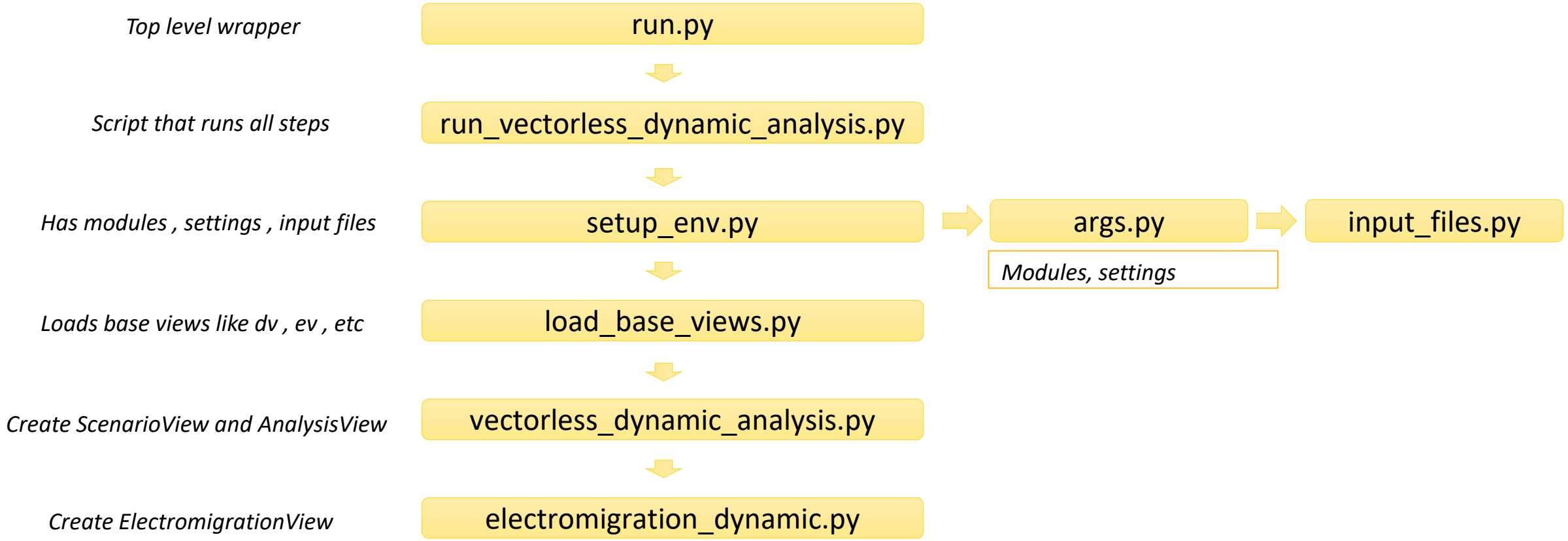
Acknowledgements:

Galaxy Testcase is created using Silvaco 45nm FreePDK libraries through Si2 OpenAccess Program

Getting Familiar with Training Lab Scripts

Lab Instructions :

- Download the Galaxy_Training.tar.gz Training Bundle
- Move to Modular_Training/04_Dynamic_ Vectorless_ Analysis directory



Getting Familiar with Training Lab Scripts

File : run.py

```
include('../scripts/run_vectorless_dynamic_analysis.py')
```

Single command file that will invoke all steps

File : scripts/run_vectorless_dynamic_analysis.py

```
include('setup_env.py')
include('load_base_views.py')
include('vectorless_dynamic_analysis.py')
Include('electromigration_dynamic.py')
```

Script runs everything from scratch and completes static , dynamic and EM analysis

Setting up the environment

File : scripts/setup_env.py

```
import pprint
open_scheduler_window()

ll = create_local_launcher('local')
register_default_launcher(ll, min_num_workers=10)

design_data_path = '../../design_data/'

include('args.py')
```

Setup for auto launching workers; here local launcher used

Set `design_data_path` variable to central design data path area

Set the arguments for various view creation commands in args.py

DataBase settings and Loading Base Views

File : scripts/load_base_views.py

```
db = open_db('../db')
# Auto-load view tags from an existing db
# Needed only for incremental(jump-start) runs
populate_view_tags()
```

Set the DB location settings using the open_db command

Auto view-tag loading for incremental (jump-start) runs

Specifying modules , arguments , settings

File : scripts/args.py

```
import package


include('./input_files.py')


options = get_default_options()
focus_pg_nets = ['VDD', 'VSS']
voltage_levels = {'VDD':1.1, 'VSS':0.0}
```

Import required Python modules

Include all design input file pointers

Include all tool options , analysis settings , etc

Specifying input files and settings

File : scripts/input_files.py

```
def_files = [
design_data_path + '/defs/Galaxy.def'
.. .. ..
]


lef_files = [
design_data_path + '/lefs/switch_cell.lef',
.. .. ..
]
```

Specify all the design files in input_files.py

File : scripts/args.py

```
dv0_args = dict(
    def_files=def_files,
    lef_files=lef_files,
    focus_pg_nets=focus_pg_nets,
    top_cell_name='Galaxy',
    tag='dv0',
    options=options)
```

Setup arguments for view creation commands in args.py

File : scripts/create_base_views.py

```
dv0 = db.create_design_view(tech_view=nv,
lib_views=lv, **dv0_args)
```

Call the arguments in view creation commands

Launching RedHawk-SC - Get started with your labs

- Batch mode execution example:
  - `<path_to_rhsc_installation>/bin/redhawk_sc galaxy.py`
- Interactive mode execution example:
  - `<path_to_rhsc_installation>/bin/redhawk_sc -i`
  - It needs an exit() command in script or entered manually to exit the Python shell
- Connecting to a live RedHawk-SC run:
  - RedHawk-SC allows querying of data/results from an active session, by remotely attaching to the session
  - Multiple users can attach to the same session from multiple machines for querying/viewing results
  - `<path_to_rhsc_installation>/bin/redhawk_sc -r <gp_dir>`
- Execution Log Files:
  - All RHSC log files reside by default in gp<> folder
    - If the run is fired in the same directory, tool will create gp.1, gp2 incrementally
    - 'latest.gp' link will point to the most recent gp directory
  - Main log file for RedHawk-SC would be <gp_directory>/run.log file.

Flows Used for Lab

- Two separate dynamic analysis done in the lab
- One with No Propagation ScenarioView
  - Toggle rate Input used here
  - Different toggle rates for top and block level
  - Referred to as No Propagation Vectorless or **NPV** flow in coming slides
- Other with Logic Propagation ScenarioView
  - Target power given for a block, i.e., PCVS
  - Referred to as **PCVS** flow in coming slides

Vectorless Analysis Settings : NPV

File : scripts/args.py

```
object_settings_npv = {
    'design_values' : {
        'clock_pin_toggle_rate': 1.6,
        'combinational_pin_toggle_rate': 0.15,
        'sequential_output_pin_toggle_rate': 0.2,
        'mode_control': {'mode_probabilities':{
                '_high_energy_mode_' : 0.3,
                '_low_energy_mode_'  : 0.4,
                '_leakage_only_mode_': 0.3}}},
    'block_values' : [
        {'pattern' : 'core3',
        'clock_pin_toggle_rate': 2.0,
        'combinational_pin_toggle_rate': 0.3,
        'sequential_output_pin_toggle_rate': 0.4,},],
    'leaf_instance_values' : [
        {'instances' : Instance("core1.program_memory"),
        'mode_control' : {'mode_sequence' : ['_leakage_only_mode_','MEM_WRITE'],},},
        {'instances' : Instance("core1.regfile_data_memory.NAND2_X1_1594"),
        'switching_control' : {'switching_sequence' : ['high','fall','low','rise']},},
        {'instances' : [Instance("HFSINV_2186_12210"),Instance("_73335_")],
        'mode_control' : {
            'mode_probabilities' : {'_high_energy_mode_' : 0.75,'_low_energy_mode_' : 0.25,}}}]}
```

Top Level Toggle Rates

Common Mode Control for All Macros

Higher Toggle Rate for block

Specific Mode Sequence for One Macro

Switching and Mode Control for some standard cells

Vectorless Analysis Settings : NPV

File : scripts/args.py

```
npv_args = dict(
    voltage_levels=voltage_levels,
    analysis_duration=32e-09,
    frame_length=8e-09,
    object_settings=object_settings_npv,
    default_clock = {'policy':'custom', 'period':8e-9},
    tag='scn_npv',
    options=options)
```

Frame length of one clock cycle

Total 4 frames

Default 8ns clock for instances with missing clock info from TWF

Vectorless Analysis Settings : PCVS

File : scripts/args.py

```
scn_activity_settings=[
    {'default_clock_period': 8e-09, 'block_name': '*', 'activity': 0.2}]

pcvs_settings = {
    'power_targets': {
        Instance('core3') : {
            '*' : {
                'target_power' : 0.080 }}},}

object_settings_pcvs = {
    'design_values' : {
        'mode_control': {
            'mode_probabilities':{
                '_high_energy_mode_' : 0.3,
                '_leakage_only_mode_': 0.7}}},
     'leaf_instance_values' : [
         {'instances' : Instance("core2.program_memory"),
         'mode_control' : {
             'mode_sequence' : 'MEM_READ','_leakage_only_mode_'],}}]}
```

User Input for Activity

Target Power for a Block

Common Mode Control for All Macros

Specific Mode Sequence for One Macro

Vectorless Analysis Settings : PCVS

File : scripts/args.py

```
clock_sta_arrival_times = {
        'clock_instance': 'sta_propagated',
        'data_instance': 'propagated',
        'sequential_launch': 'propagated',
        'non_sequential_start_point': 'sta_propagated',}

sta_transition_times = {
        'clock_instance': 'sta_propagated',
        'data_instance': 'sta_propagated',
        'sequential_launch': 'sta_propagated',
        'non_sequential_start_point': 'sta_propagated',}
```

STA arrival times for clock and propagated times for data instances

All transition times to be picked from STA file

Vectorless Analysis Settings : PCVS

File : scripts/args.py

```
scn_pcvs_args = dict(
    voltage_levels=voltage_levels,
    scenario_duration=48e-9,
    activity_level = scn_activity_settings,
    target_power_settings = pcvs_settings,
    event_time_precedence = clock_sta_arrival_times,
    transition_time_precedence = sta_transition_times,
    object_settings = object_settings_pcvs,
    tag='scn_pcvs',
    options=options)
```

48ns i.e. 6 cycles of analysis

Analysis View Settings

File : scripts/args.py

```
av_dynamic_pcvs_args = dict(
    duration=40e-09,
    step_size=20e-12,
    tag='av_dynamic_pcvs',
    scheduler_barrier =False,
    options=options)

av_dynamic_npv_args = dict(
    duration=24e-09,
    step_size=20e-12,
    tag='av_dynamic_npv',
    scheduler_barrier =False,
    options=options
```

Scheduler barrier turned off to allow two AVs to run in parallel

Reporting Utilities

File : scripts/vectorless_dynamic_analysis.py

```
power_utils.report_block_power(scn_npv,file_name=reports_dir+'/npv_power_sum
mary.rpt',float_format='{v:.2e}')

emir_reports.write_all_instance_voltages(av_dynamic_npv,
reports_dir+'/inst_voltage_npv.rpt')
emir_reports.write_bump_currents(av_dynamic_npv,
reports_dir+'/bump_current_npv.rpt')
emir_reports.write_bump_voltages(av_dynamic_npv,
reports_dir+'/bump_voltage_npv.rpt')
emir_reports.write_demand_currents(av_dynamic_npv,
reports_dir+'/demand_current_npv.rpt')
emir_reports.write_layer_voltage_report(av_dynamic_npv,
reports_dir+'/layer_voltage_npv.rpt')
```

Shown only for NPV Scenario and it's AnalysisView

Electromigration Analysis from Dynamic Currents

File : scripts/args.py

```
pem_rms_args = dict(
    tag='pem_rms',
    mode='rms',
    delta_t_rms=10,
    options=options)

pem_peak_args = dict(
    tag='pem_peak',
    mode='peak',
    options=options)
```

File : scripts/electromigration_dynamic.py

```
pem_rms = db.create_electromigration_view(av_dynamic_npv, **pem_rms_args)
pem_peak = db.create_electromigration_view(av_dynamic_npv, **pem_peak_args)

reports_dir = 'reports/powerem_analysis'
gp_util.makedir(reports_dir)
for em_type,em_view in { 'RMS_EM' : pem_rms , 'PEAK_EM' : pem_peak
}.items():
    emir_reports.write_em_metal_report(em_view,
reports_dir+"/"+em_type+'_metal_report.rpt')
    emir_reports.write_em_via_report(em_view,
reports_dir+"/"+em_type+'_via_report.rpt'
```

RMS and Peak EM done for PG nets
Only NPV flow considered here

Log File Review

File : latest.gp/run.log

```
 16     WARNING<CSG.104> Some instance voltages are out of range of library data: $text
 441    WARNING<CSG.106> Using 0.0 for leakage of Instance($instance_id)
Pin('$pin_name') Cell('$cell_name') since no when conditions matched and cell has no
unconditional leakage specified.
 201    WARNING<STR.406> Instance pin '$input_string' not found in design ($file:$line)
18     WARNING<SCN.123> STA const pin does not exist: Instance($instance_id)
Cell('$cell_name') Pin($pin_id) $msg
 9      WARNING<SCN.127> Instance($instance_id) Pin('$pin_name')$cell_type is not
connected to a net during logic propagation. Assuming constant HIGH
 1      WARNING<SCN.164> Insufficient duration to cover all the modes given in
mode_control $type for Instance($instance_id)
 8      WARNING<SCN.169> No current table with input pin present for
Instance($instance), using input pin Pin($pin) to create events
1100   WARNING<NPV.105> Instance($inst_id) does not have a valid timing window on output
pin Pin($pin_id). This instance would not switch in the scenario.
 1809   WARNING<NPV.107> Instance($inst_id) does not have a valid connected output pin.
This instance would not switch in the scenario.
```

CSG : Current Waveform Generation

STR : STA file reading

SCN : Logic Prop Scenario

NPV : NPV Scenario

Log File Review : PCVS info

File : latest.gp/run.log

```
INFO<PCV.120>       Frame size: 7.99999977374e-09, number of frames: 6, scenario_duration: 4.8e-08.
INFO<PCV.119>       Non scalable power is 0.0148377863942 and maximum power is 0.694767987231.
```

Can search for 'PCV' in log

Auto calculated frame size

Non scalable power is the minimum power i.e. leakage power and clock instances not covered by any ICG

Maximum power is by turning on all ICGs and toggling all sequential instances

Switching Summary Report

File : reports/pcvs_switching_coverage.rpt

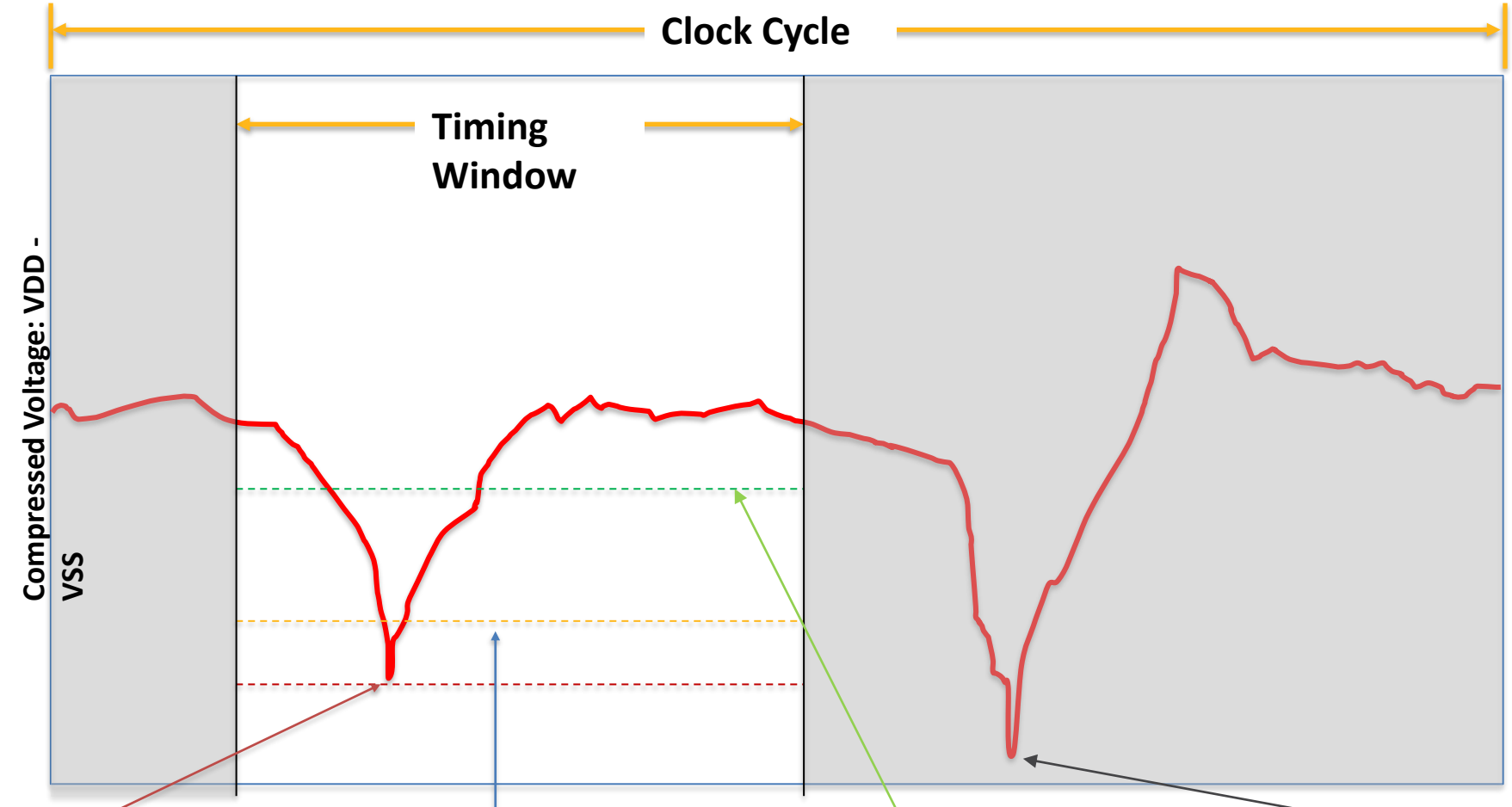
```
Switching Coverage for Block: '' Domain: 'core3/VDD_INT'
Cell Type Wise Coverage for Frame: 32.00ns - 40.00ns
+------------------+-------------+-------------+---------------+-------------+---------------+
|    Cell Type     |    Total    |  Switching  |  Switching %  |  Cumulative |  Cumulative % |
+------------------+-------------+-------------+---------------+-------------+---------------+
|      Clock       |     149     |     138     |    92.62 %    |     138     |    92.62 %    |
|       ICG        |      68     |      2      |     2.94 %    |      2      |     2.94 %    |
|    Sequential    |    17950    |     5859    |    32.64 %    |    14309    |    79.72 %    |
|  Combinational   |    76626    |    20816    |    27.17 %    |    49522    |    64.63 %    |
| ---------------- | ----------- | ----------- | ------------- | ----------- | ------------- |
|      Total       |    94793    |    26815    |    28.29 %    |    63971    |    67.48 %    |
+------------------+-------------+-------------+---------------+-------------+---------------+

Cell Type Wise Coverage for Frame: 40.00ns - 48.00ns
+------------------+-------------+-------------+---------------+-------------+---------------+
|    Cell Type     |    Total    |  Switching  |  Switching %  |  Cumulative |  Cumulative % |
+------------------+-------------+-------------+---------------+-------------+---------------+
|      Clock       |     149     |     138     |    92.62 %    |     138     |    92.62 %    |
|       ICG        |      68     |      2      |     2.94 %    |      2      |     2.94 %    |
|    Sequential    |    17950    |     5974    |    33.28 %    |    15128    |    84.28 %    |
|  Combinational   |    76626    |    22463    |    29.32 %    |    52778    |    68.88 %    |
| ---------------- | ----------- | ----------- | ------------- | ----------- | ------------- |
|      Total       |    94793    |    28577    |    30.15 %    |    68046    |    71.78 %    |
+------------------+-------------+-------------+---------------+-------------+---------------+
```

Per frame switching numbers

PCVS given for the block core3

Dynamic Voltage Drop Reporting: Parameters



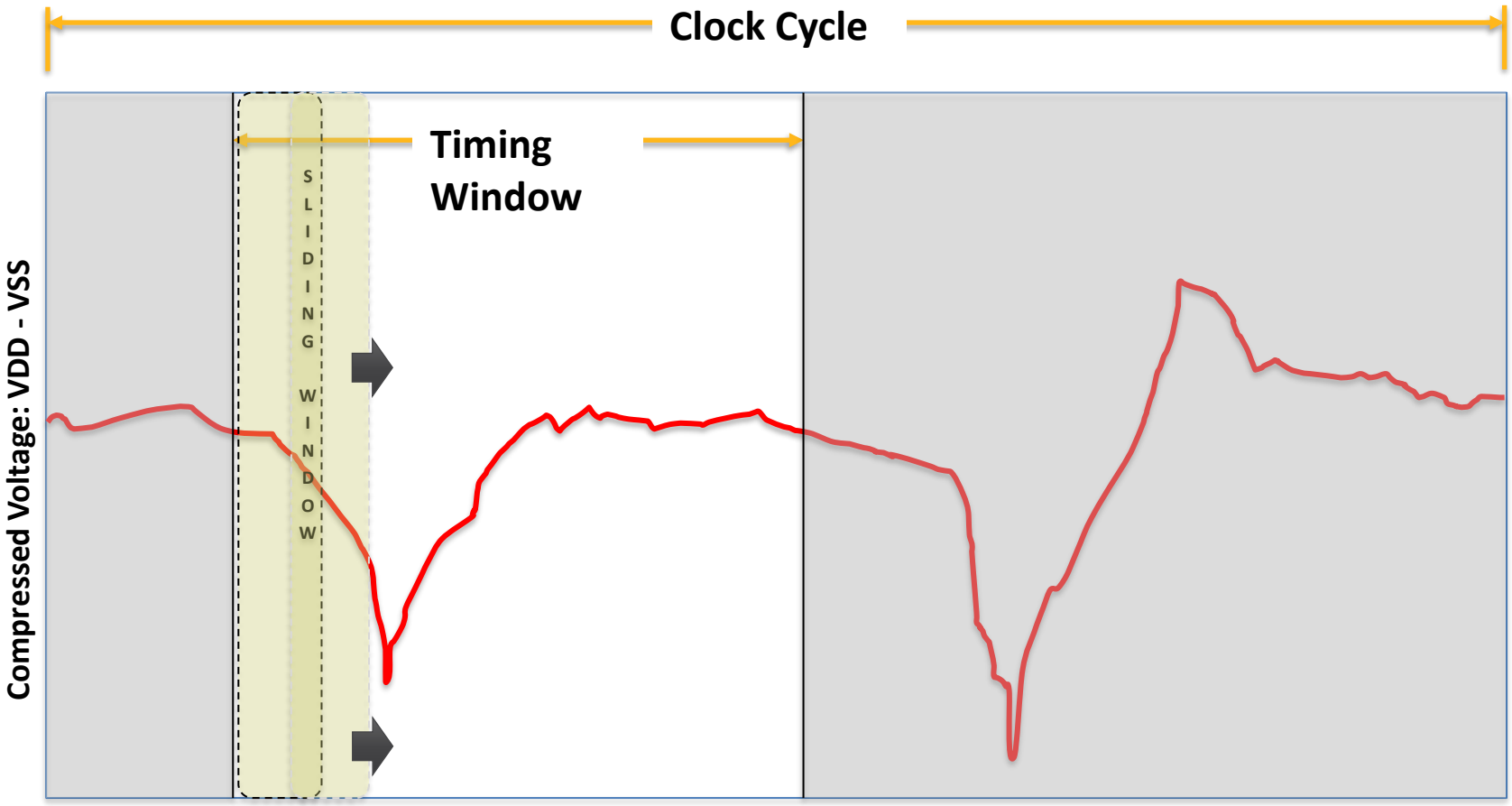
minTW: Min within timing window (pessimistic)

avgTW: worst average across sliding window (see next slide)

Average within timing window (optimistic)

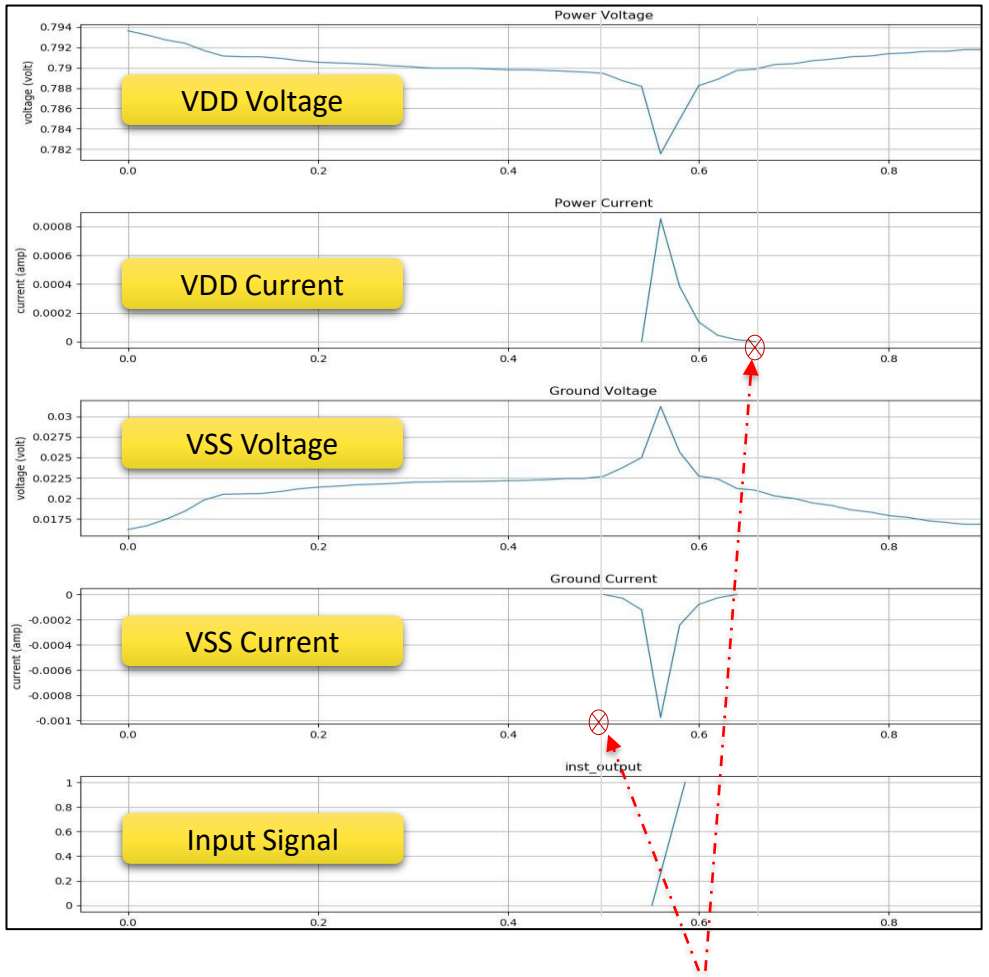
minWC: Worst across the cycle; can be outside timing window

Moving Average Approach (Sliding Window)



- A sliding window is created representing the switching duration (time when instance is sensitive to voltage drop)
- This window is slowly moved across the Timing window and the average voltage drop (across sliding window) is measured
- **AvgTW** is measured as the worst among sliding window average voltage drop values

Effective DVD measurement



Parameter	Description
Effective DVD	Average VDD-VSS voltage in a current stamping window
Rise effdvd	Effdvd when any of the output signals rises
Fall effdvd	Effdvd when any of the output signals falls
in_only effdvd	Effdvd when there is no transition on output signals
fullsim effdvd	Minimum of the Effdvd for each type across all cycles
fullsim avgTW	Minimum of the cycle AvgTW rise/fall data across all cycles

"Effective DvD Window": overlap window of currents of all PG pins of instance

Dynamic Voltage Drop Reporting

```
TitleText: Bump Currents
# Time (ps) I (A)
"VSS_1
      0.00   -0.0000005
     20.00   -0.0000804
     40.00   -0.0002011
     60.00   -0.0003409
     80.00   -0.0005351
    100.00   -0.0008380
```

bump_current_npv.rpt

```
TitleText: Bump Voltages
# Time (ps) V (V)
"VSS_1
      0.00    0.0000002
     20.00    0.0003660
     40.00    0.0011909
     60.00    0.0029072
     80.00    0.0060383
    100.00    0.0085382
```

bump_voltage_pcvs.rpt

```
TitleText: Demand Currents
# Time (ps) I (A)

"VSS
      0.00   -0.1367024
     20.00   -0.1656296
     20.00   -0.1656296
     20.00   -0.1656296
     20.00   -0.1656296
     20.00   -0.1656297
     20.00   -0.1656297
     40.00   -0.1997199
     40.00   -0.1997199
     40.00   -0.1997199
```

```
# loc_x    loc_y    eff_Vdd   max_pg_tw  min_pg_tw  min_pg_sim  max_pg_sim  min_vdd_tw  max_vss_tw   pg_arc      instance
#  (u)      (u)      (v)       (v)        (v)        (v)         (v)         (v)         (v)      pwr/gnd
 1075.615  484.505   1.054     1.1        0.9762     0.961       1.113       0.9763     8.716e-05  core3/VDD_INT/VSS core3/cts_inv_527224759
 1082.455  484.505   1.099     1.104      1.099      0.9621      1.113       1.1        0.0003504  core3/VDD_INT/VSS core3/HFSINV_46_15798
 1083.215  484.505   1.092     1.113      1.083      0.9623      1.113       1.089      0.006373   core3/VDD_INT/VSS core3/regfile_program_memory.AOI21_X1_923
 1080.365  481.705   1.096     1.113      1.094      0.9641      1.113       1.096      0.00238    core3/VDD_INT/VSS core3/regfile_program_memory.NOR2_X1_978
 1082.265  481.705    1.1      1.1        1.097      0.9649      1.113       1.098      0.000928   core3/VDD_INT/VSS core3/HFSINV_31_13751
 1082.645  481.705   1.085     1.1        1.093      0.965       1.113       1.095      0.001559   core3/VDD_INT/VSS core3/HFSINV_31_16038
 1083.215  481.705   1.099     1.1        1.099      0.9651      1.113       1.1        0.0008513  core3/VDD_INT/VSS core3/HFSINV_40_14888
```

inst_voltage_npv.rpt

```
TitleText: Demand Currents
# Time (ps) I (A)

"core3/VDD_INT
      0.00    0.0054279
     40.00    0.0051139
     80.00    0.0051307
    100.00    0.0058439
    120.00    0.0084662
    140.00    0.0090562
    140.00    0.0090562
    160.00    0.0078603
    200.00    0.0052586
    220.00    0.0053284
```

demand_current_npv.rpt

```
#  min_x     min_y    min_voltage_drop   max_x     max_y    max_voltage_drop  layer_drop  net        layer
#   (u)       (u)          (v)            (u)       (u)          (v)            (v)
 660.885   432.645      0.01067      1081.78    483.165       0.1299         0.1192    core3/VDD_INT metal1
 1226.355   569.8       0.006487      176.96    1057.0       0.07906        0.07257    VSS          metal1
 1226.07     0.0        0.008334     836.085    1136.8       0.07319        0.06486    VDD          metal1
 1175.14    628.6       0.006742      31.615    1194.2       0.03627        0.02952    VSS          metal2
 1174.94    628.6       0.006764      31.74     1194.2       0.03549        0.02873    VSS          metal3
```

layer_voltage_pcvs.rpt

Power EM Reporting

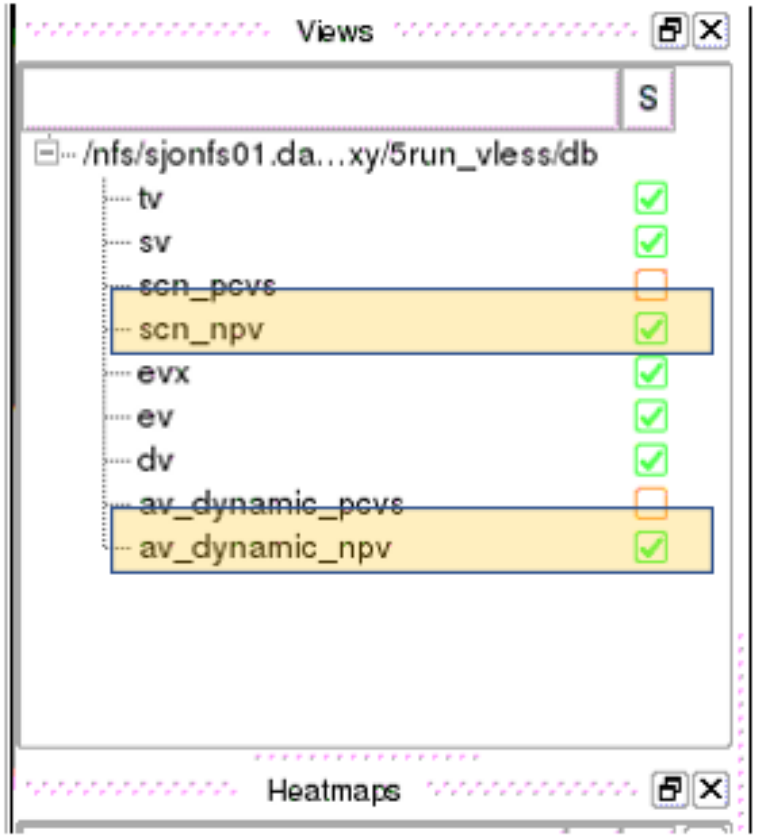
```
# em_type = PEAK, ignore_sliver = True, 'length' column is blech length
#  layer          from                to             length     width      current   constraint violation  status net
#                  (u)                 (u)             (u)      (u)        (A)       (A)       (%)
  metal1    (678.0775,420.105)    (678.25,420.105)     153.3      0.17    0.01111    0.002179    509.7    FAIL  VSS
  metal1    (678.605,420.105)     (678.615,420.105)    153.3      0.17     0.0111    0.002179    509.3    FAIL  VSS
  metal1     (677.46,420.105)      (677.84,420.105)    153.3      0.17    0.01041    0.002179    477.8    FAIL  VSS
  metal1     (677.31,420.105)      (677.46,420.105)    153.3      0.17    0.009715   0.002179    445.8    FAIL  VSS
  metal1    (1075.89,543.305)     (1076.09,543.305)    295.1      0.17    0.009612   0.002179    441.1    FAIL  VSS
  metal1    (1076.09,543.305)     (1076.255,543.305)   295.1      0.17     0.0096    0.002179    440.6    FAIL  VSS
  metal1    (1076.945,543.305)    (1077.065,543.305)   295.1      0.17    0.009594   0.002179    440.3    FAIL  VSS
  metal1    (1077.065,543.305)    (1077.6,543.305)     295.1      0.17    0.009589   0.002179    440      FAIL  VSS
  metal1    (1077.825,543.305)    (1078.3675,543.305)  295.1      0.17    0.009581   0.002179    439.7    FAIL  VSS
```

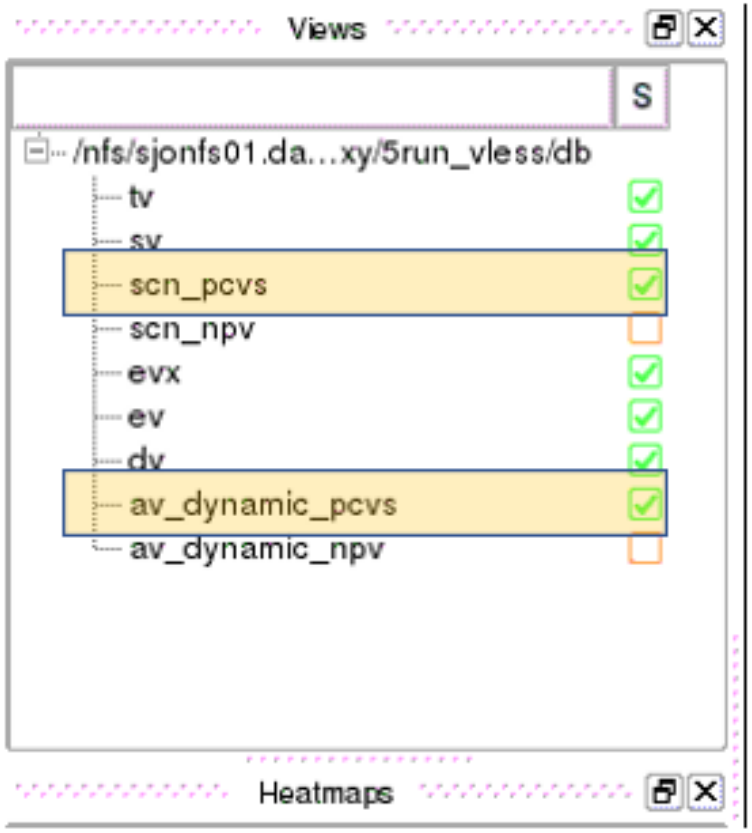
powerem_analysis/PEAK_EM_metal_report.rp

```
# em_type = RMS, ignore_sliver = True
# layer      loc_x       loc_y     via_length  via_width  current   constraint violation  status net
#             (u)         (u)         (u)         (u)      (A)        (u)       (%)
```

powerem_analysis/RMS_EM_via_report.rpt

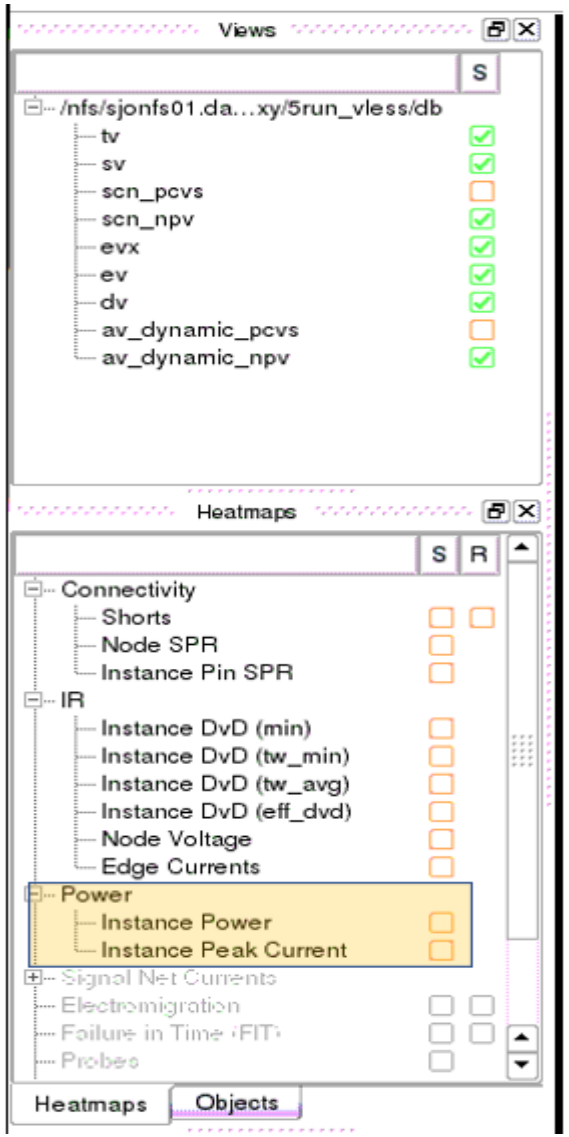
GUI

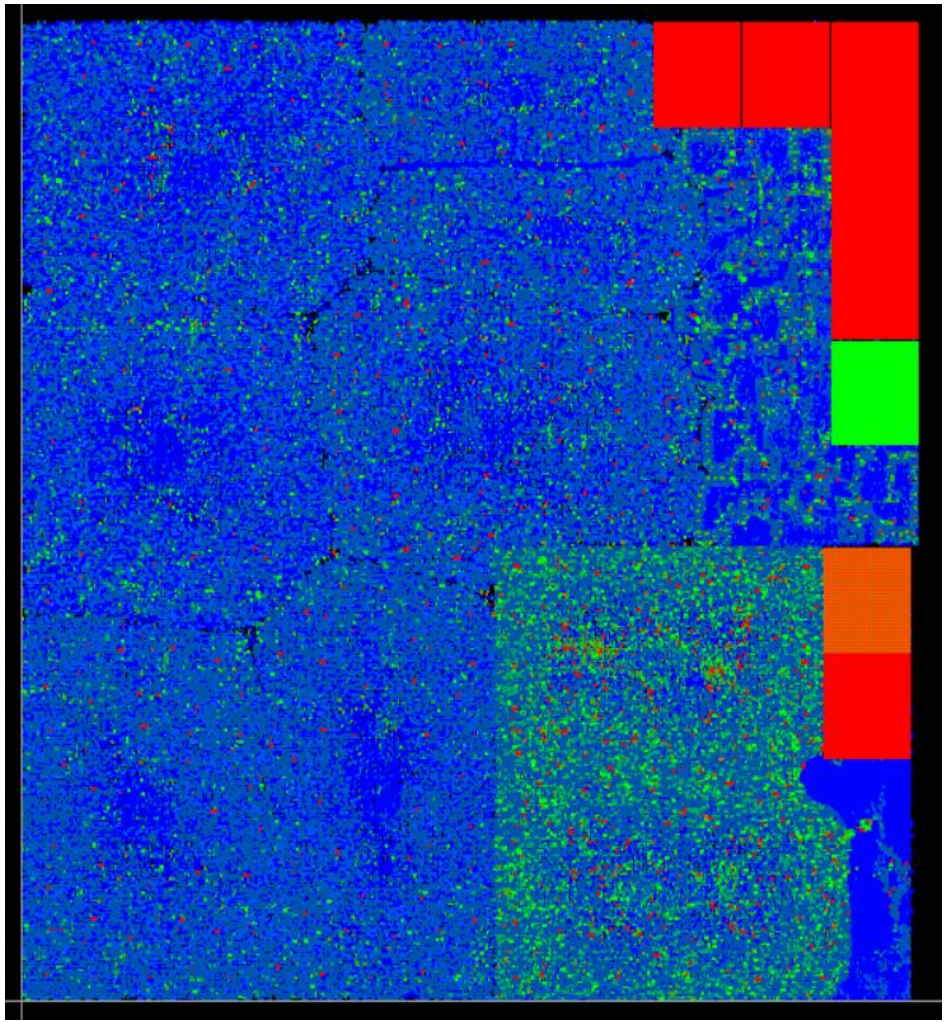




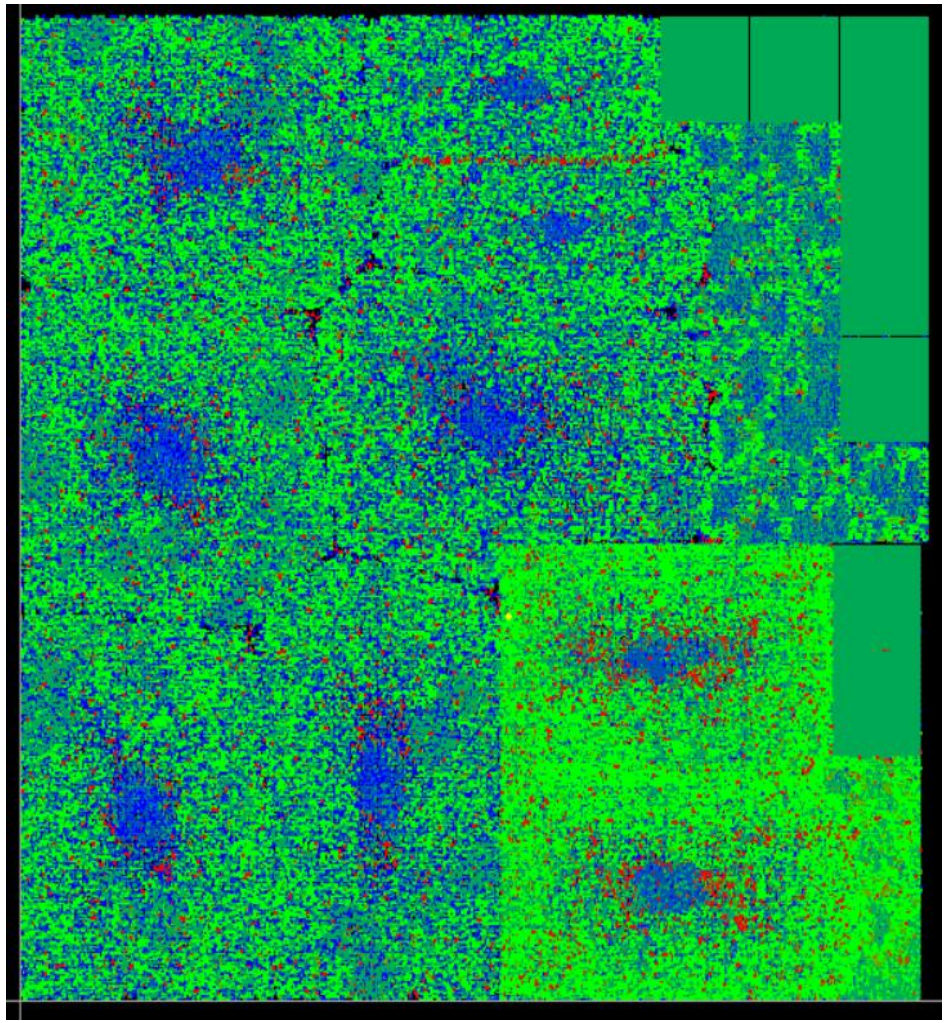
Ensure that only the right views are selected when multiple SCN/AV is present

Power/Current Heatmaps



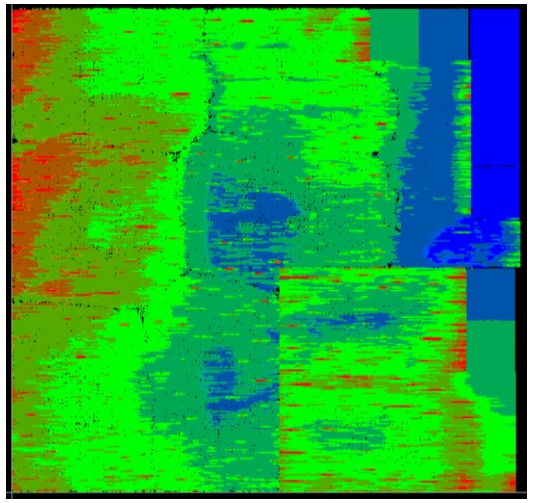


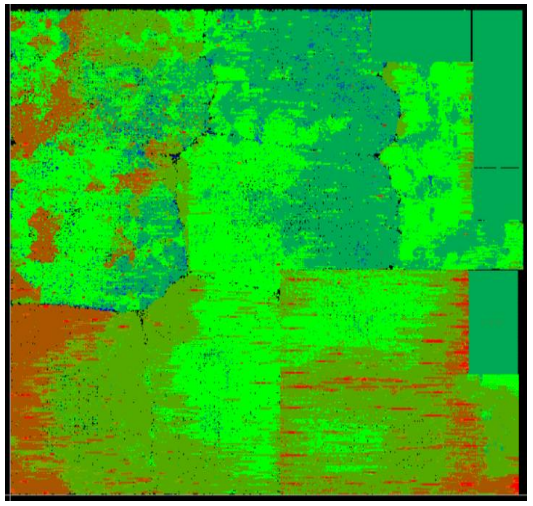
Instance Power Map
Logic Prop SCN PCVS

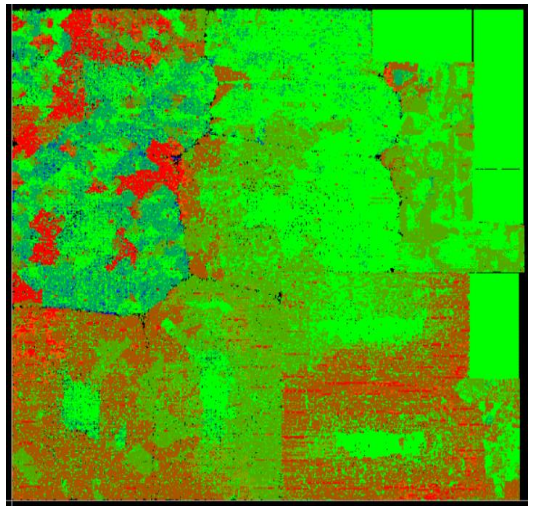


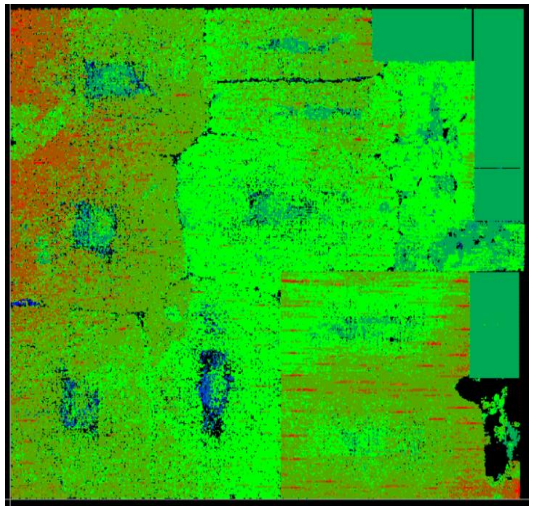
Instance Peak Current Map
NPV

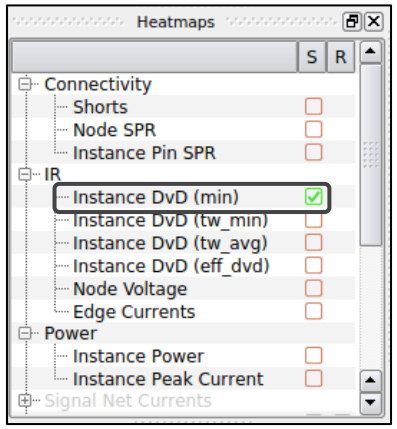
Instance Voltage Drop Parameters



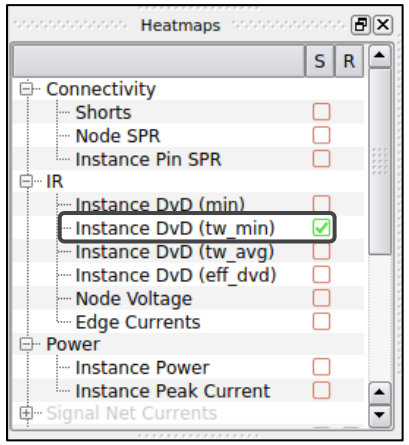




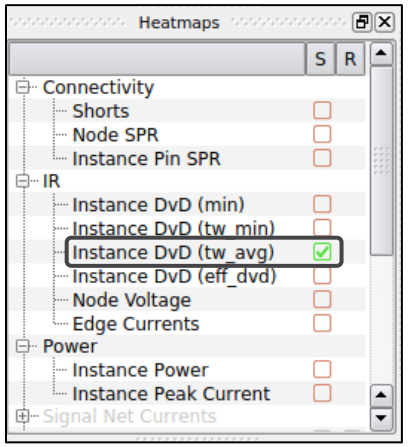




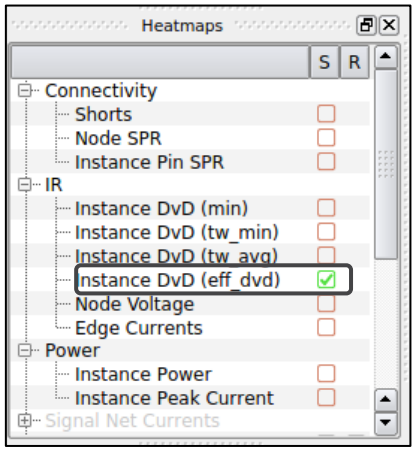
Min DVD compression Over Whole Cycle



Min DVD compression Over Timing Window



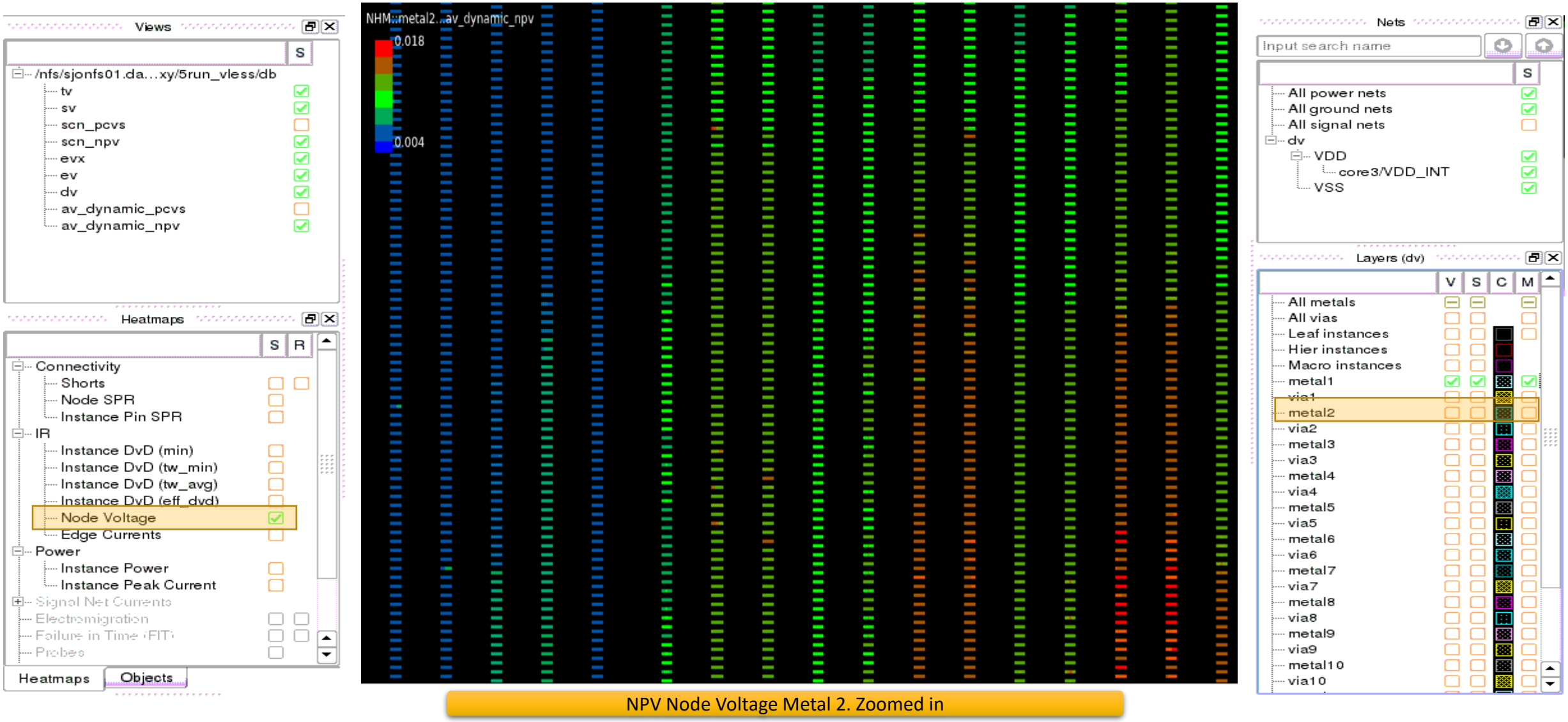
Average DVD compression Over Timing Window



Effective DVD compression Measured over switching window

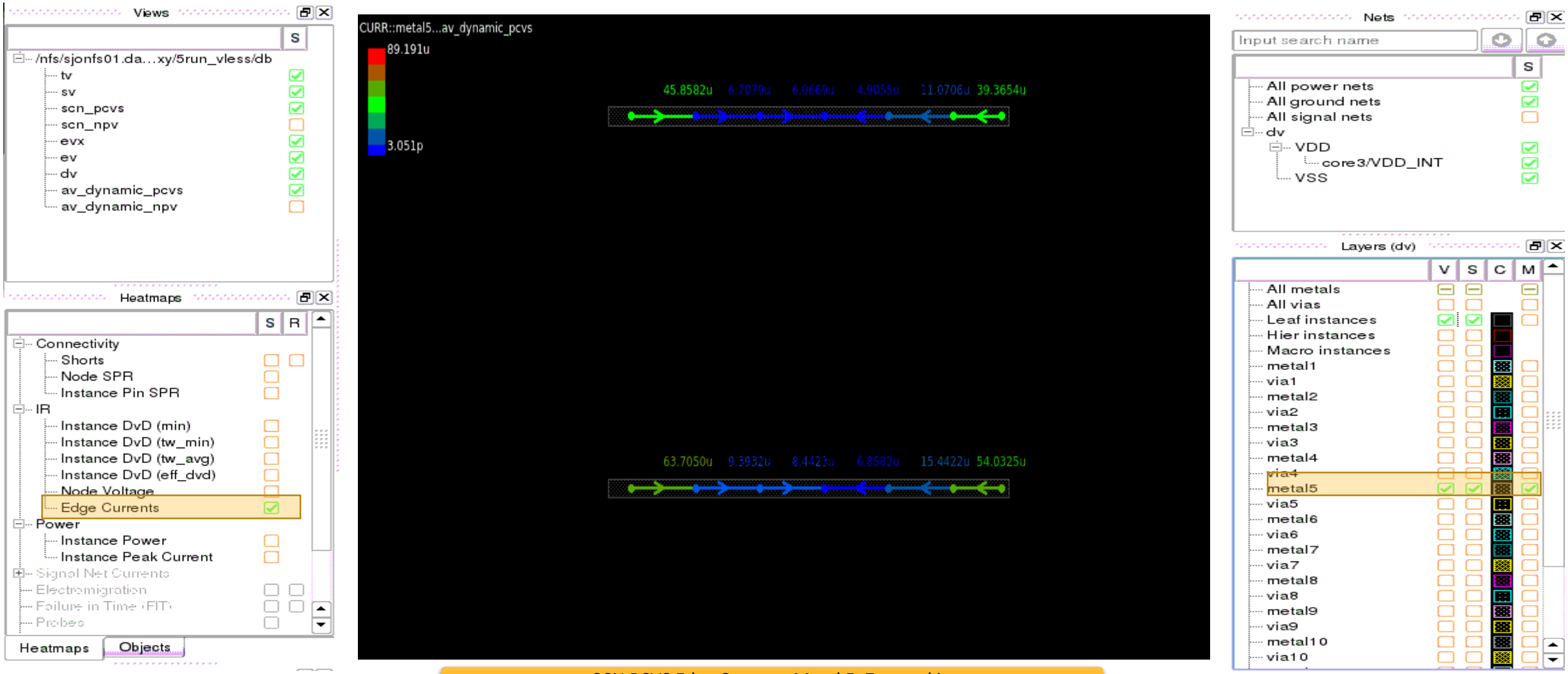
All snaps from Logic Prop SCN PCVS Scenario

Node Voltages



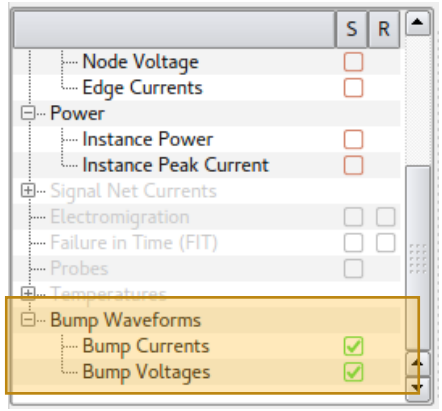
NPV Node Voltage Metal 2. Zoomed in

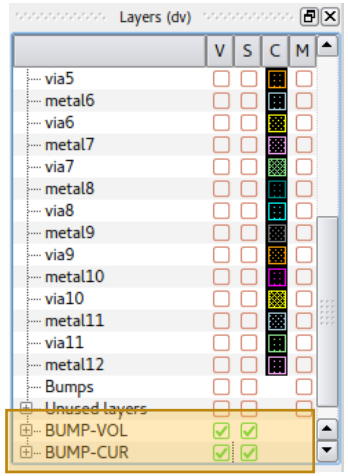
Edge Currents

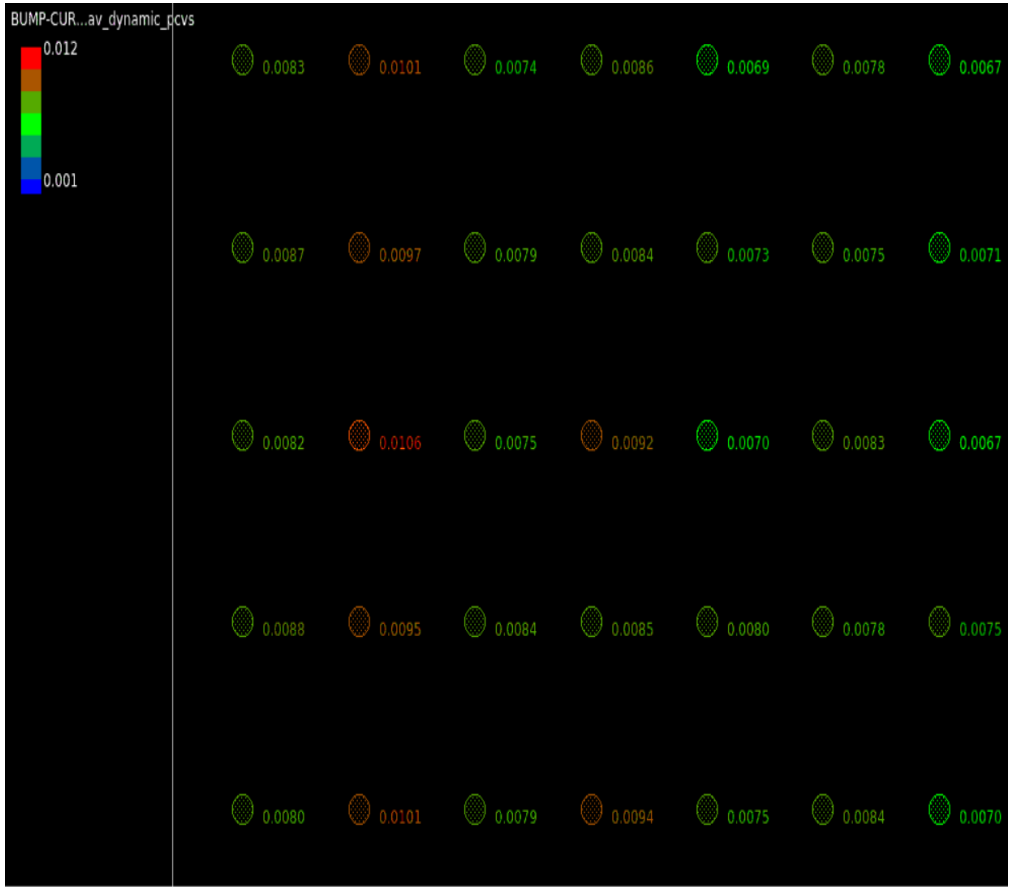


SCN PCVS Edge Currents Metal 5. Zoomed in

Bump Current/Voltage Maps





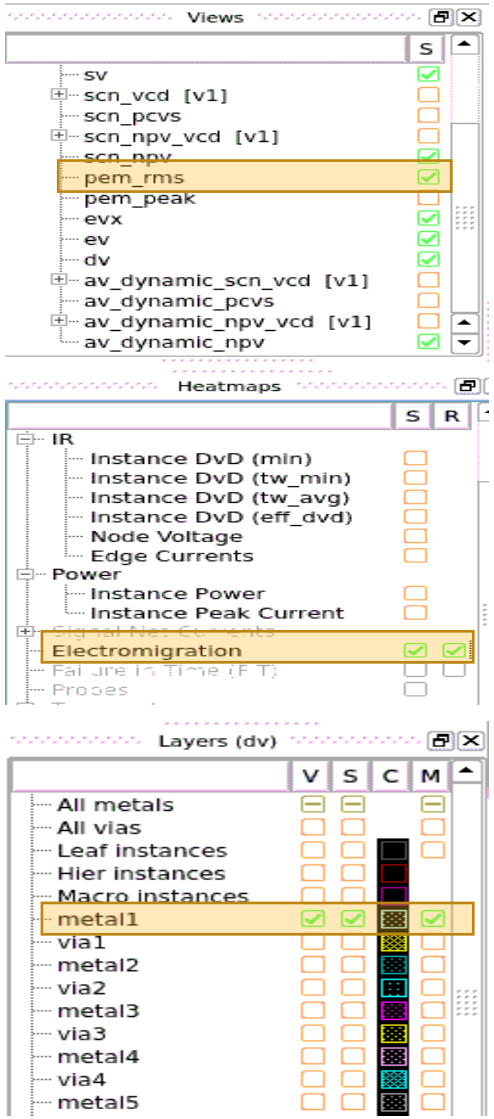


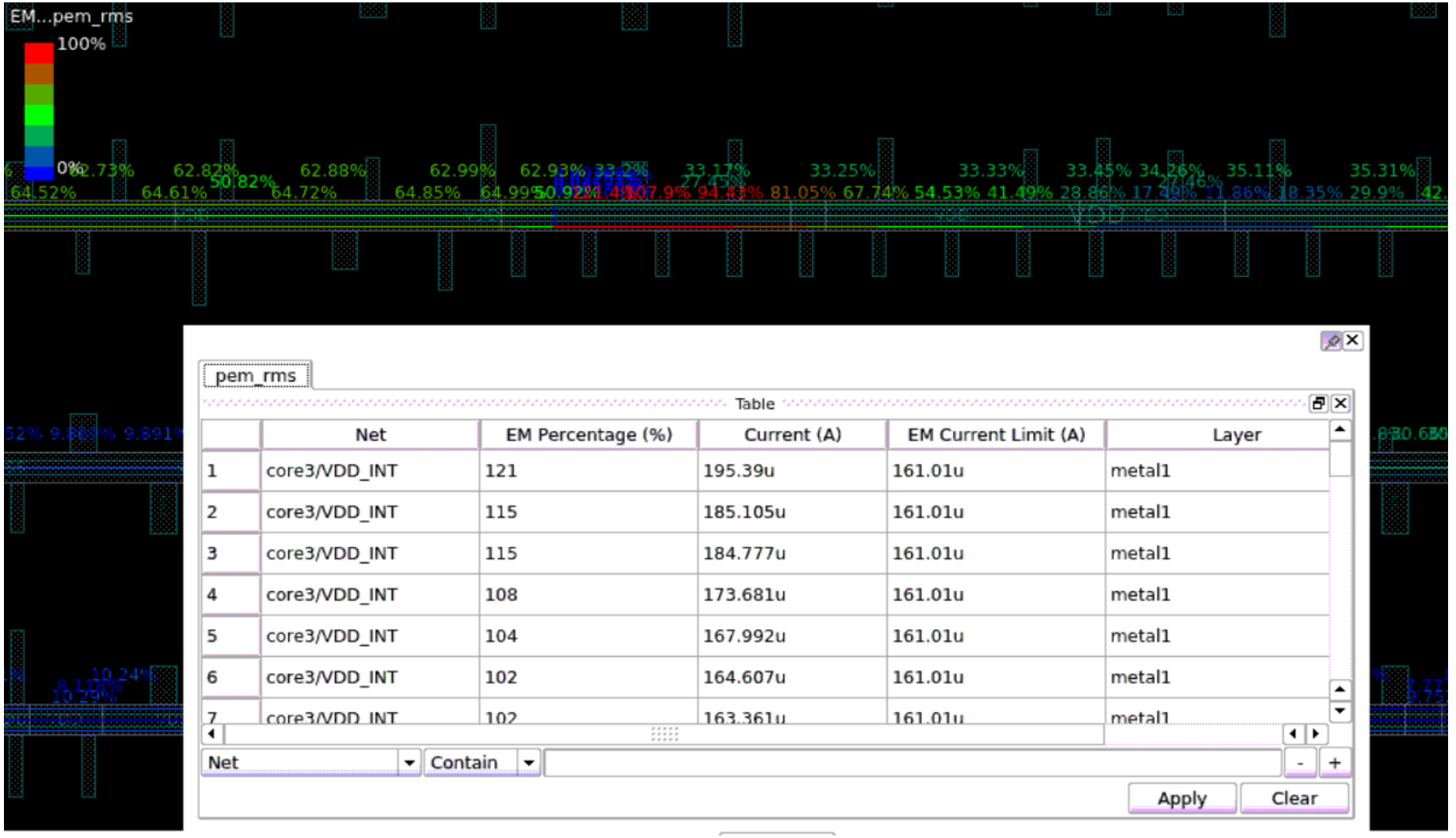
Bump Voltage Map

Bump Current Map

Both from SCN PCVS Scenario

Electromigration Analysis Maps RMS EM

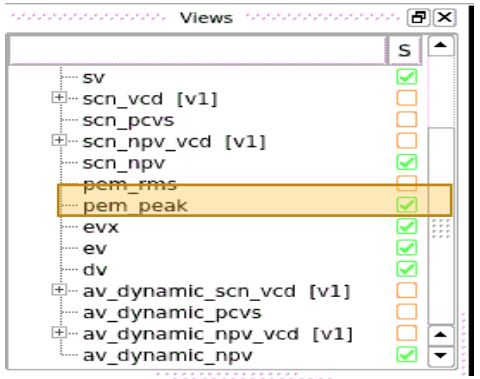


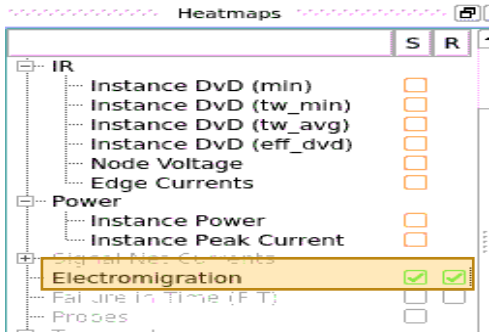


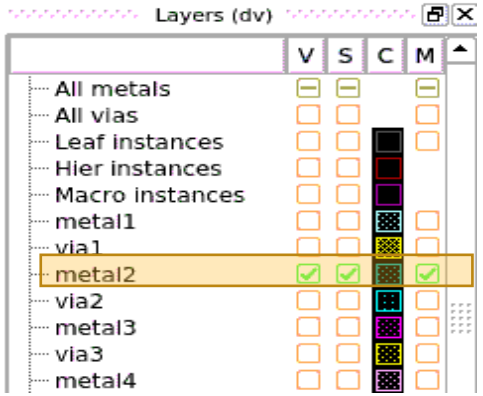
Metal1 RMS EM Heatmap

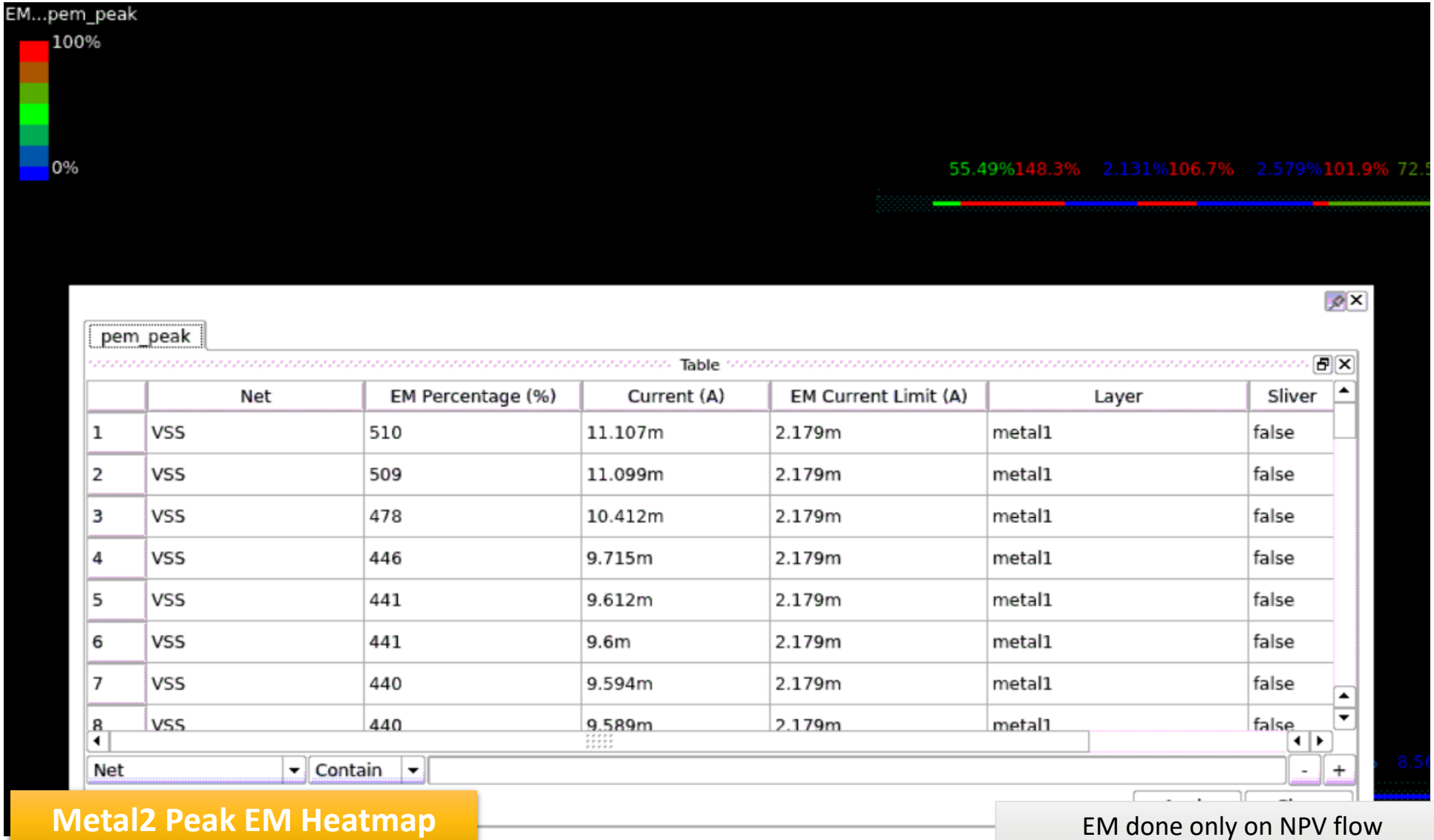
EM done only on NPV flow

Electromigration Analysis Maps Peak EM





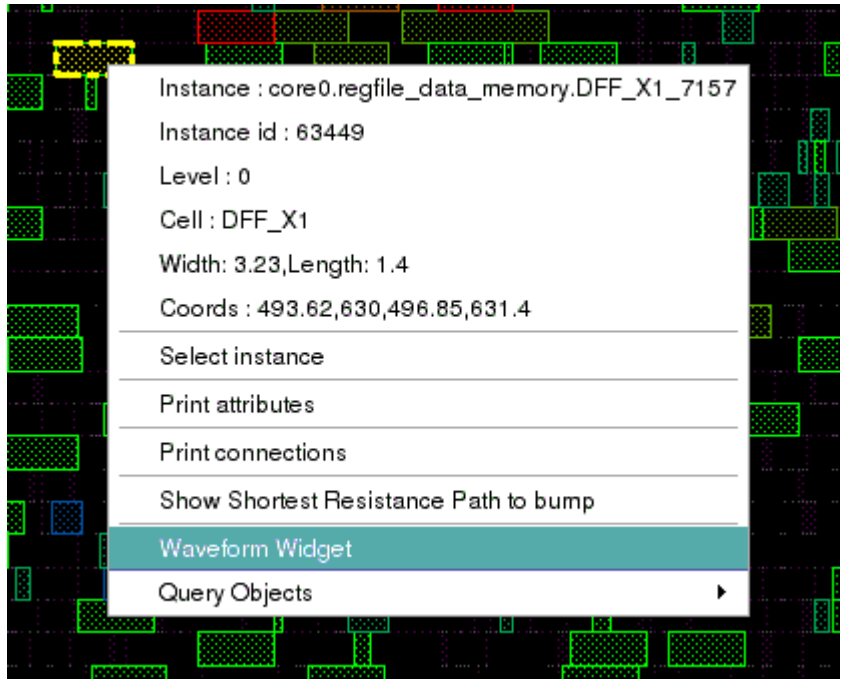


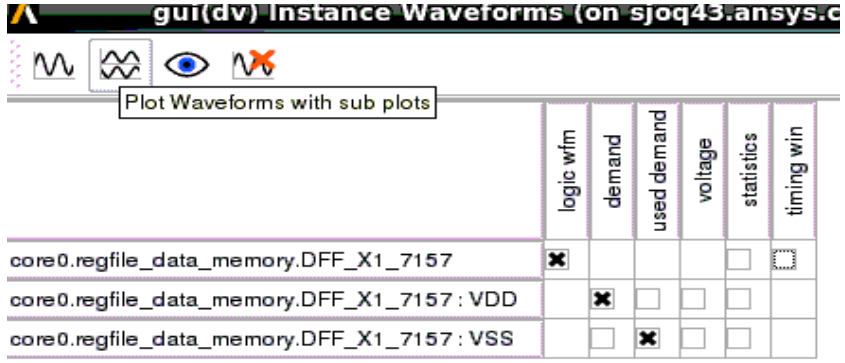


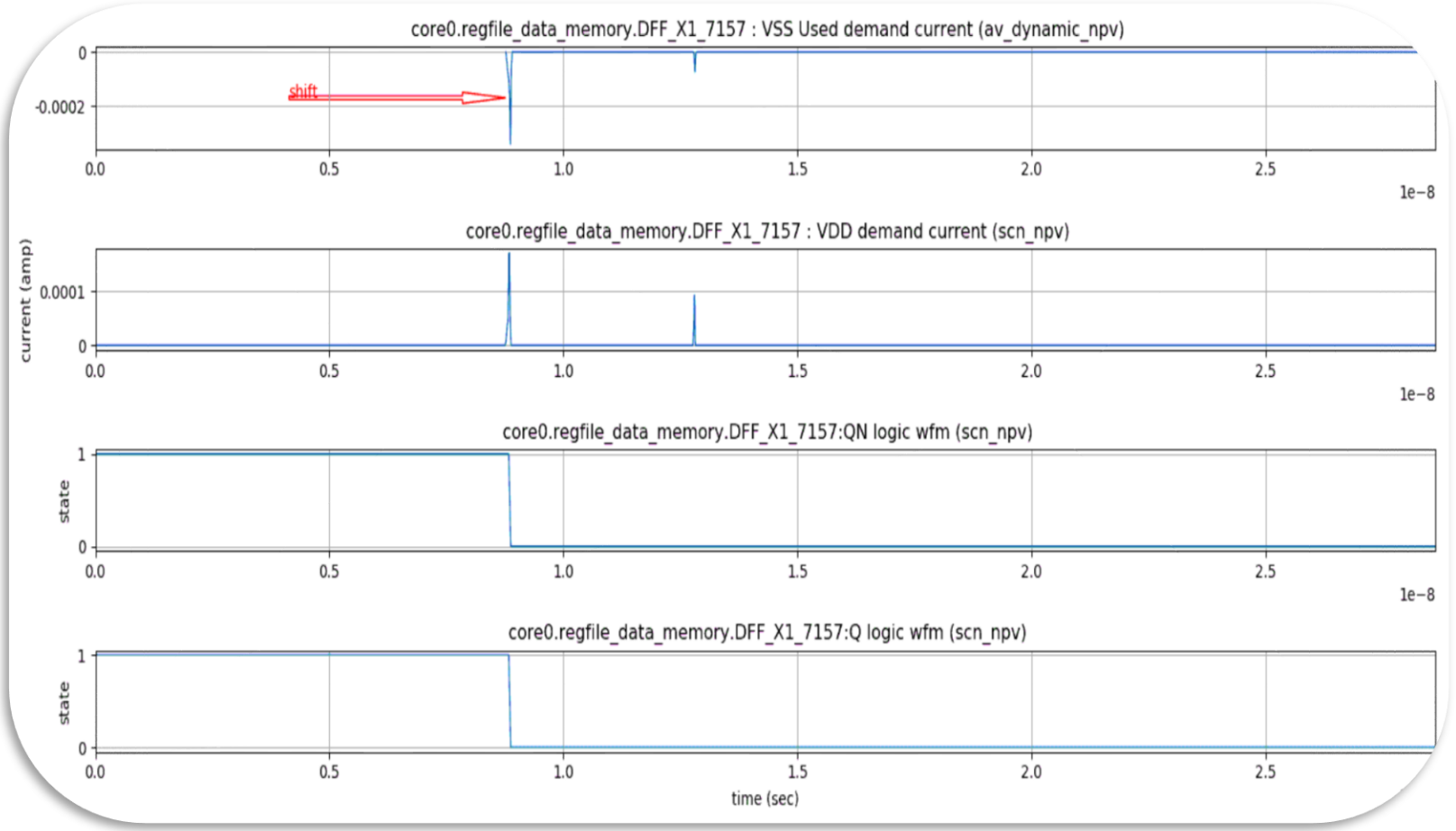
Metal2 Peak EM Heatmap

EM done only on NPV flow

Instance level voltage/current/logic waveforms







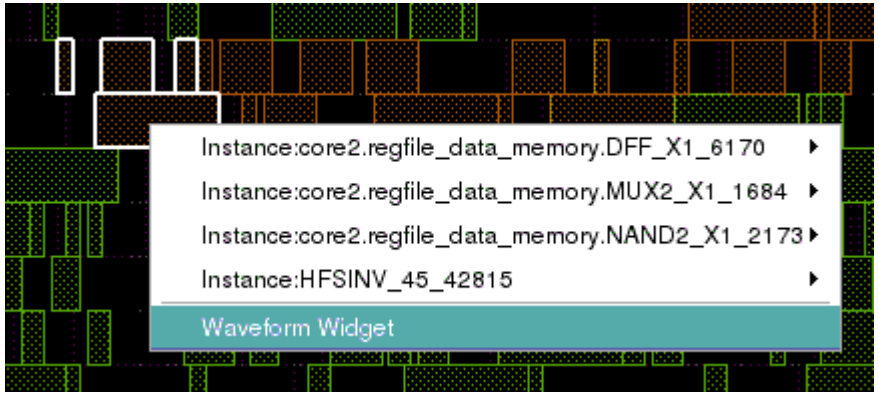
From NPV Scenario

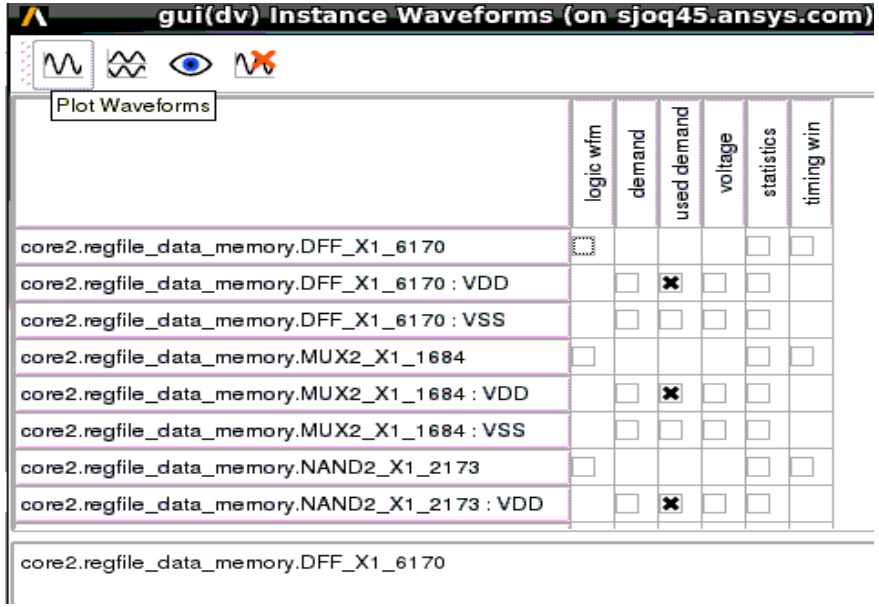
Instance level voltage/current waveforms : Care abouts

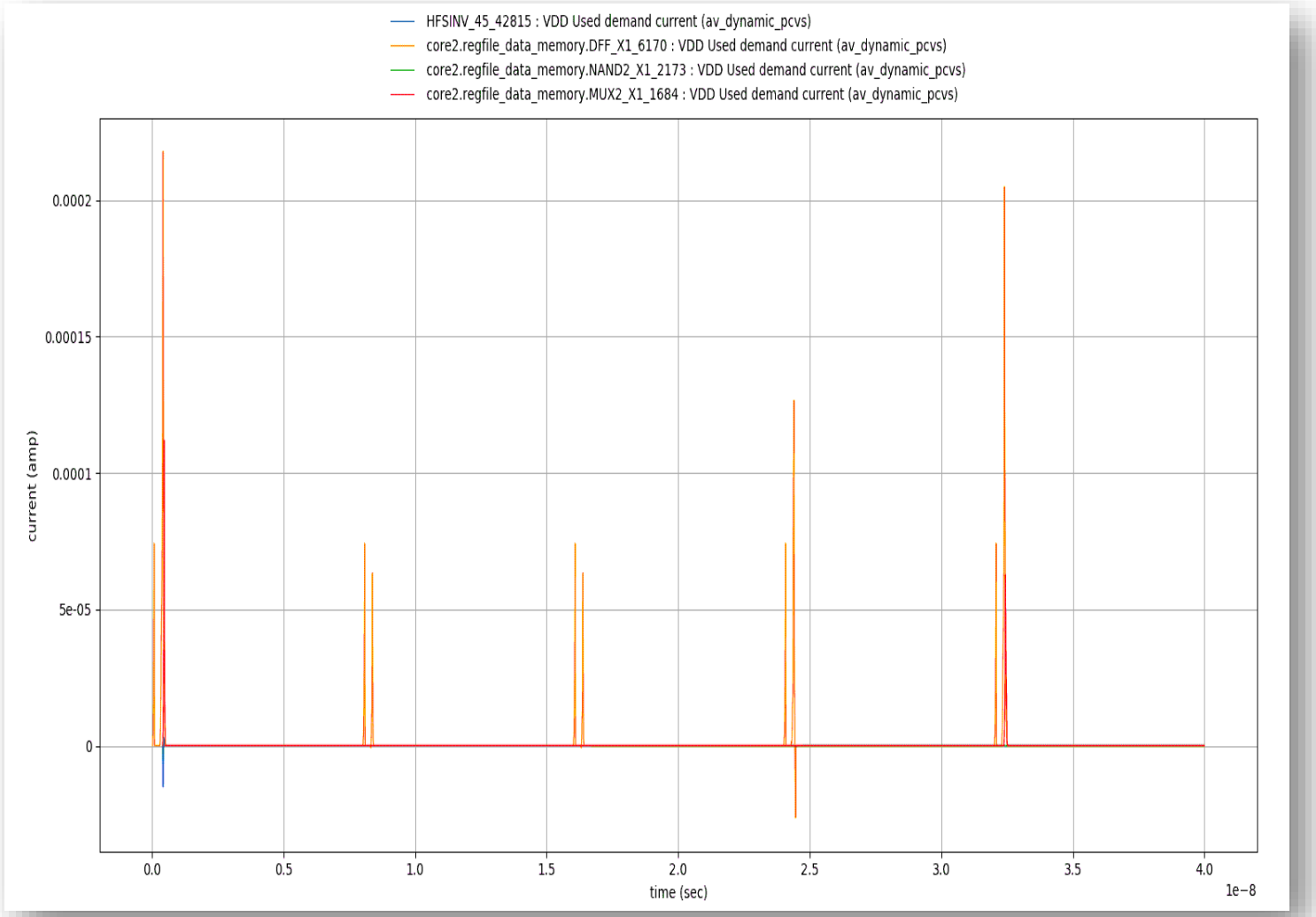
- For instance voltage waveforms , must have `keep_stats_level='Full'` in `create_analysis_view`
  - Default is "Medium" -> provide voltage stats like minTW, effTW, etc but doesn't save waveforms
  - Saving waveforms in 'Full' level will consume disk space
- 'Demand Current' from Scenario View , 'Used Demand Current' from AnalysisView.
  - 'Used Demand Current' is sampled by analysis time step , 'Demand Current' is raw PWL without sampling
  - For achieving quick stability before dynamic simulation starts, demand current will be shifted to average value.
- Logical waveform of only output pins available directly from GUI
  - There are APIs to examine events for all pins

Combined Waveform of Multiple Instances

Select Multiple Instances

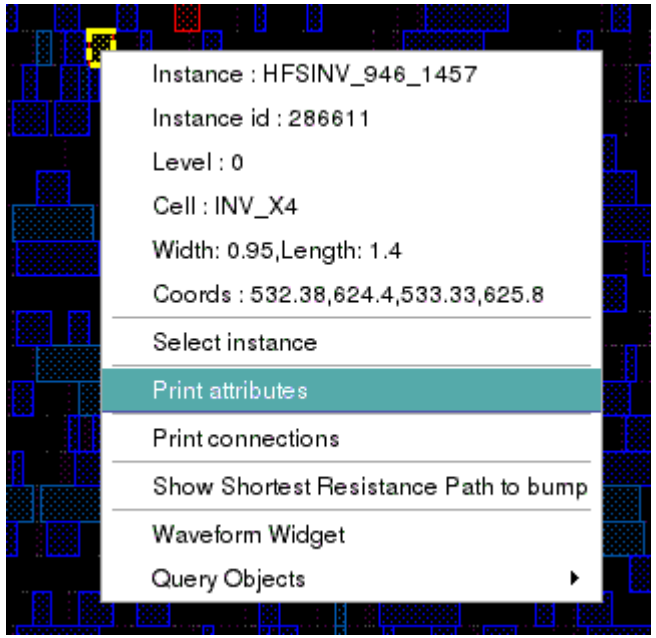


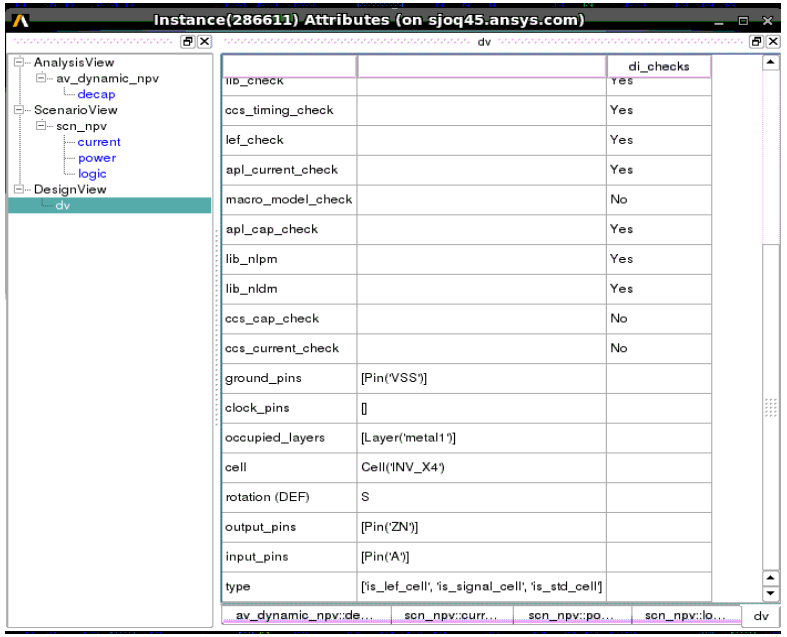


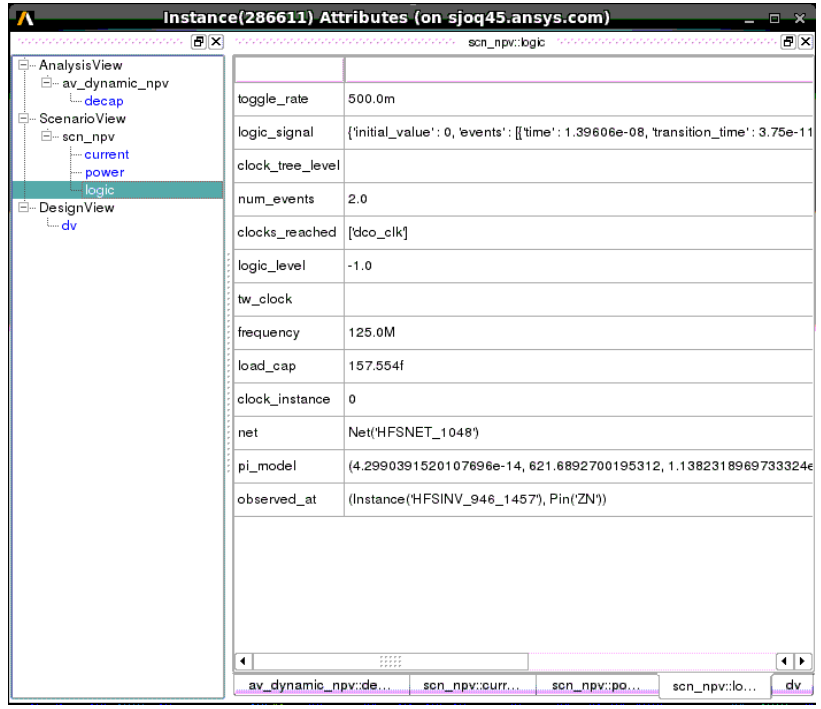


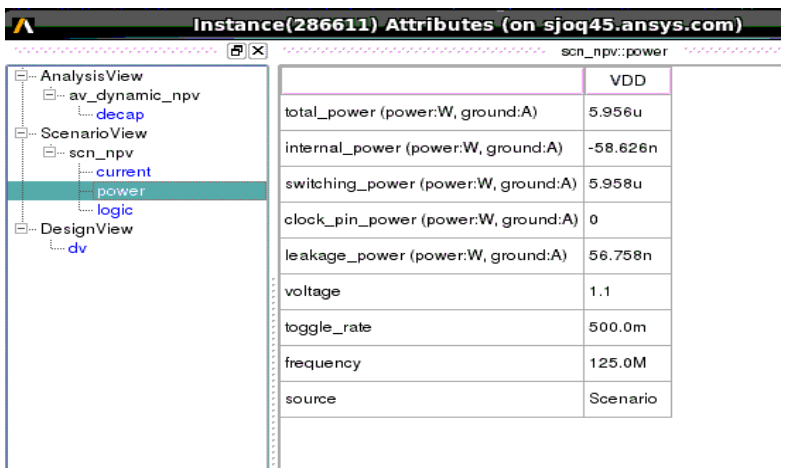
They switch close together. Taken from SCN PCVS Scenario

Attributes from GUI









Interactive Shell

```
%redhawk_sc --console
```

Brings up console GUI as well as an interactive shell
DBs and views loaded via console will be available in the interactive shell

```
%redhawk_sc -i
>>> gp.open_db('./db')
>>> gp.populate_view_tags()
```

Bring up just the shell
Load db
Command to auto load all available views

Design Level Current/Voltage Waveforms

```
>>> scn_npv.get_total_demand_currents().keys()
[Net('core3/VDD_INT'), Net('VSS'), Net('VDD')]
>>> plot(scn_npv.get_total_demand_currents()[Net('VDD')])
```

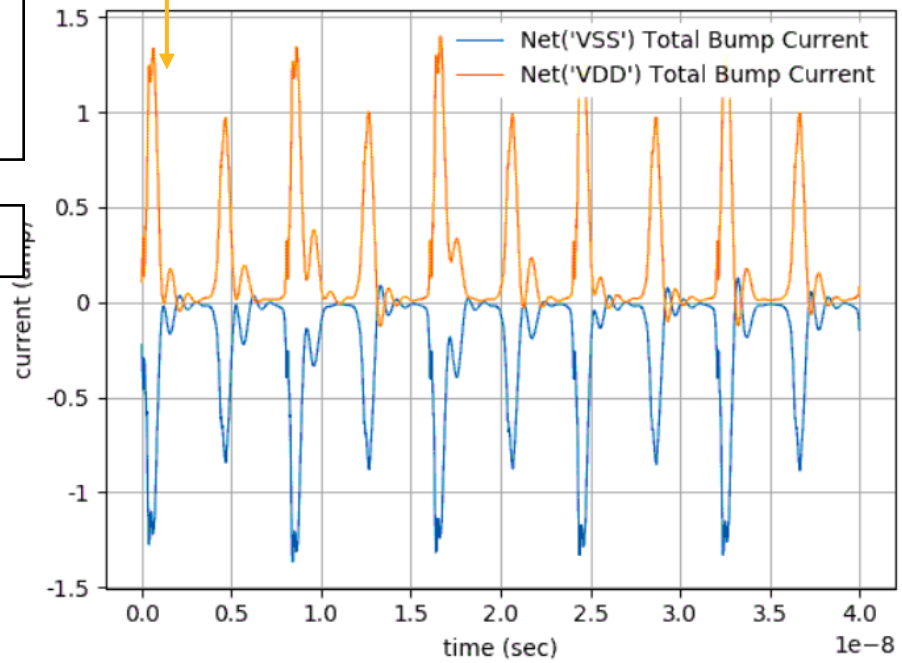
```
plot(av_dynamic_pcvs.get_total_used_demand_currents()
```

```
>>> av_dynamic_npv.get_average_bump_voltages().keys()
[Net('VSS'), Net('VDD')]
>>> plot(av_dynamic_npv.get_average_bump_voltages()[Net('VDD')])
```

```
plot(av_dynamic_pcvs.get_total_bump_currents())
```

`plot` command allows a lot of flexibilities. `help` had details

Can view all domains together as here or separate as given above



Voltage Drop Histogram

```
>>> help(av_dynamic_npv.get_node_voltage_histograms)
get_node_voltage_histograms()
  Returns histograms of voltage drop for all nets and
layers (nodes) in the power / ground grid.
>>> nvh = av_dynamic_npv.get_node_voltage_histograms()
>>> plot(nvh[Net('VDD')][Layer('metal1')])
```

```
>>> dvd_histogram =
av_dynamic_pcvs.get_instance_voltage_histogram()
>>> plot(dvd_histogram,hist_type='bar',log=False,)
```

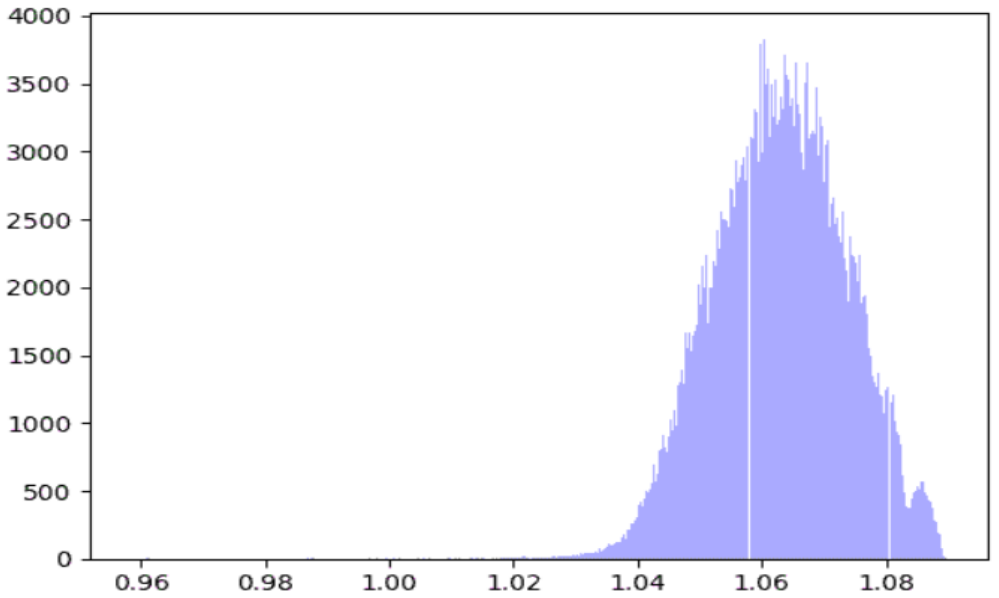
Some modifications done to default plot

Histograms are a quick way to understand about overall voltage drop picture. Worst drop or GUI could be misleading

Per domain and per layer voltage values

Refer to help for info on Histogram bin settings

Voltage numbers seen across all instances
Default parameter is global minimum.
Can be set to desired parameter



Individual Instance Details

- Important APIs. Some of them explained in coming slides
- Checking help for each command will return usage details

API	Output
tv.get_attributes()	Transition time, arrival time, clock root
scn.get_attributes()	Logic propagation details and events, power and current details
scn.get_mode_usage_attributes()	For macro, modes that got applied
dv.cell_mode_attributes()	Macro master cell, high/low/median mode details
scn.get_logic_signals()	Logic signal for each signal pin
scn.get_instance_events()	Event arcs considered for each instance
scn.get_demand_current()	Demand current waveform
av.get_voltage_stats()	Voltage drop parameters for the cell
av.get_voltage()	Voltage waveform. Stored if keep_stats is kept as Full
av.get_bump_voltages()	Bump voltage waveform
av.get_current()	Bump current waveform

Instance's Logic Details

```
>>> scn_pcvs.get_attributes(Instance("core1.regfile_program_memory.OAI21_X1_845"))
{'logic': {'clock_instance': False,
         'clock_tree_level': None,
         'clocks_reached': ['dco_clk'],
         'frequency': 125000000.0,
         'load_cap': 6.97111571484714e-14,
         'logic_level': 10,
         'logic_signal': {'initial_value' : 0, 'events' : [{'time' : 1.39186e-08,
'transition_time' : 2.175e-10, 'dir' : 'r'}, {'time' : 2.09964e-08, 'transition_time' :
1.275e-10, 'dir' : 'f'}, {'time' : 4.50762e-08, 'transition_time' : 2.175e-10, 'dir' :
'r'}, ],},
         'net': Net('core1.regfile_program_memory._13887_'),
         'num_events': 3,
         'observed_at': (Instance('core1.regfile_program_memory.OAI21_X1_845'),
Pin('ZN')),
         'pi_model': (1.112773784936474e-16, 321.74737548828125, 6.955248416052814e-14),
         'toggle_rate': 0.5,
         'tw_file': {'tw_clock': 'dco_clk', 'tw_max': 6.196999802199343e-09, 'tw_min':
4.998999880712063e-09}},
'power': {Pin('VDD'): {'clock_pin_power': 0.0,……..
```

Clock or Data Instance
Clock Root and Frequency

Logic Level : How deep in combo network

Number of events and toggle rate for output pin

Instance's Events

```
>>> scn_npv.get_instance_events(Instance("core1.regfile_program_memory.DFF_X1_5510"))
{0: [{'events': [{'CK': 'f',
                  'Q': '0',
                  'input_event_time': 2.490399886312389e-08,
                  'input_transition_time': 5.999999802552836e-11},
                  {'CK': 'r',
                  'Q': 'r',
                  'input_event_time': 2.8903997417728533e-08,
                  'input_transition_time': 6.499999843923021e-11}],
   'net_cap': 5.047869979272427e-15,
   'out_pin': 'Q',
   'total_cap': 7.538433988369189e-15},
   {'events': [{'CK': 'f',
                  'QN': '1',
                  'input_event_time': 2.490399886312389e-08,
                  'input_transition_time': 5.999999802552836e-11},
                  {'CK': 'r',
                  'QN': 'f',
                  'input_event_time': 2.8903997417728533e-08,
                  'input_transition_time': 6.499999843923021e-11}],
   'net_cap': 1.5443945109842758e-16,
   'out_pin': 'QN',
   'total_cap': 1.5443945109842758e-16}]}
```

Demand Current Created for the given transition time, load cap, input to output pin combination

Macro Modes Used

```
>>> scn_npv.get_mode_usage_attributes(Instance("core3/program_memory"))
{Pin('clk0'): {Pin('vdd'): [(4.6815000764866e-09, 'MEM_READ'),
                            (1.2681500294320358e-08, 'MEM_READ'),
                            (2.0681499179886487e-08, '_leakage_only_mode_'),
                            (2.8681498065452615e-08, 'MEM_WRITE')]}}
```

Details of mode applied each clock cycle

```
>>> dv.get_cell_mode_attributes(Cell('sram_16_512_freepdk45'))
{'default_process': {
    'apl': {(1.0, 25.0): {},(1.100000023841858, 25.0): {Pin('vdd'): {0: {
        '_high_energy_mode_': ('MEM_WRITE', '1'),
        '_low_energy_mode_': ('MEM_READ', '1'),
        '_median_energy_mode_': ('MEM_WRITE', '1')}}}},
    'ccsp': {(1.0, 25.0): {}, (1.100000023841858, 25.0): {}},
    'nlpm': {(1.0, 25.0): {Pin(''): {61: {
        '_high_energy_mode_': ('', '!csb0 * !clk0 * web0'),
        '_low_energy_mode_': ('', 'csb0'),
        '_median_energy_mode_': ('', '!csb0 * clk0 * !web0')}}},
          (1.100000023841858, 25.0): {}}}}
```

NLPM Boolean state or APL modes corresponding to high/median/low can be understood

Voltage Stats

```
>>> av_dynamic_npv.get_voltage_stats(Instance('core2.regfile_data_memory.DFF_X1_2013'))
full_sim: Min: 1.08176 Max: 1.11267 Mean: 1.09976 Count: 1201 vss_at_min_:0.00670747
full_sim_over_tw: Min: 1.0825 Max: 1.09686 Mean: 1.08915 Count: 39 vss_at_min_:0.00825669
full_sim_effdvd: Min: 0 Max: 0 Mean: 0 Count: 0 vss_at_min_:0
```

The output that gets printed is a short summary

```
>>> vs = av_dynamic_npv.get_voltage_stats(Instance('core2.regfile_data_memory.DFF_X1_2013>
>>> type(vs)
<class 'gp_export_py.SMInstanceArcVoltageStats'>
>>> help(vs)
FUNCTIONS
get_cycle_effdvd()
  Get effective dvd statistics for one cycle of the simulation.

get_cycle_over_tw()
  Get statistics for one cycle of the simulation within timing window.
…….
```

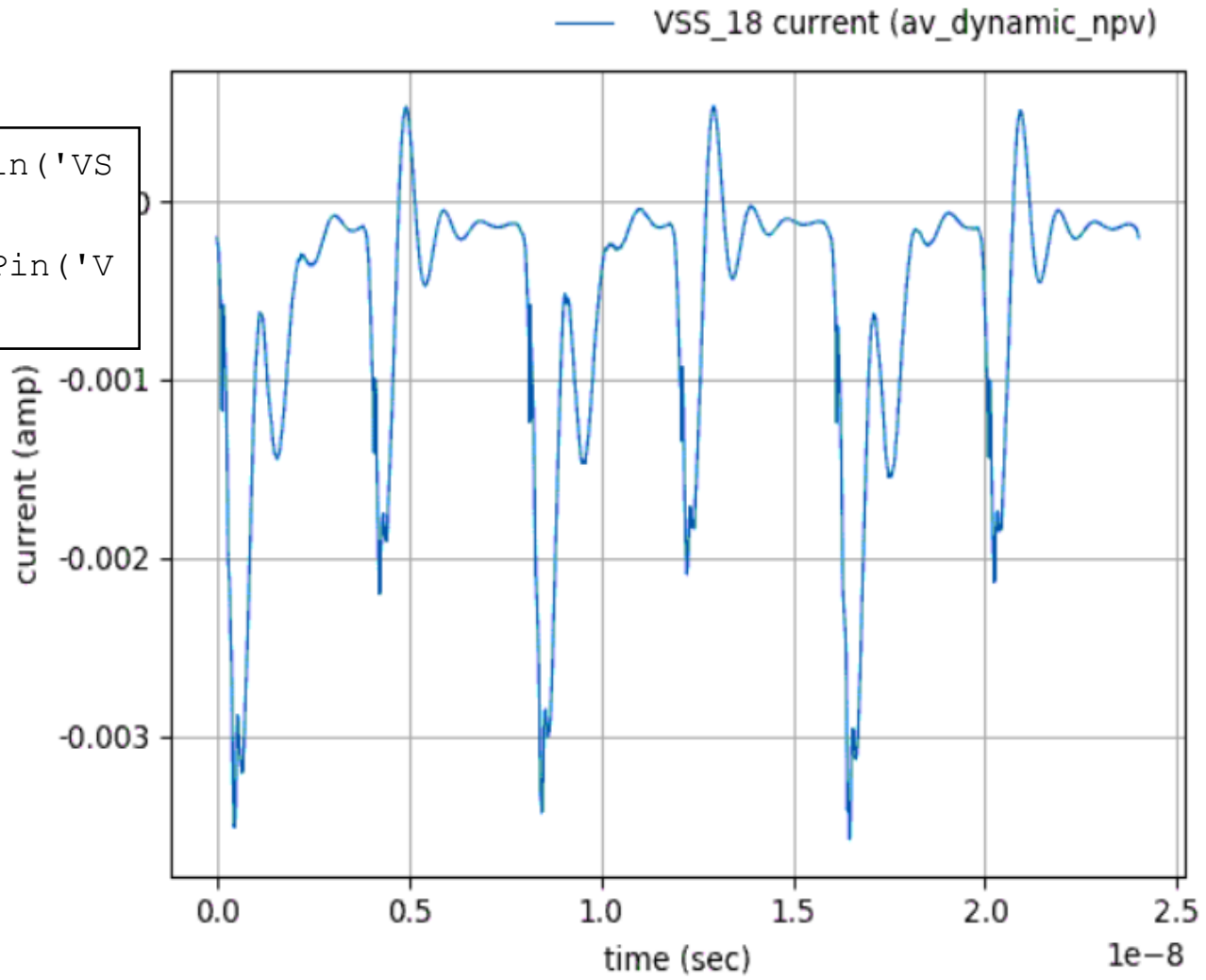
An object is returned which gives much more info

Bump Currents

The API is for per instance current waveform
Can we be used for retrieving bump currents

```
plot(av_dynamic_npv.get_current(Instance(''),Pin('VSS_18')),legend_location='top right')
plot(av_dynamic_pcvs.get_current(Instance(''),Pin('VDD_17')))
```

Instance('') means top design. Its pins would be the bumps in the design.



Questions

and

Answers

Thank You


Ansys