ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση Κατανάλωσης Ισχύος Ολοκληρωμένων Κυκλωμάτων

Power Analysis of Integrated Circuits

Διπλωματική Εργασία

Αναστάσιος Θ. Μελισσόπουλος

Επιβλέποντες Καθηγητές:	Γεώργιος Σταμούλης Καθηγητής
	Νέστωρας Ευμορφόπουλος Επίκουρος καθηγητής

Βόλος, Σεπτέμβριος 2018

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση Κατανάλωσης Ισχύος Ολοκληρωμένων Κυκλωμάτων

Διπλωματική Εργασία

Αναστάσιος Θ. Μελισσόπουλος

Επιβλέποντες καθηγητές: Γεώργιος Σταμούλης
Καθηγητής

Νέστωρας Ευμορφόπουλος
Επίκουρος Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 2018

....	....
Γ. Σταμούλης	Ν. Ευμορφόπουλος
Καθηγητής	Επίκουρος Καθηγητής

……………………………..

Αναστάσιος Μελισσόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Πανεπιστημίου Θεσσαλίας

To my family and friends

Στην οικογένειά μου και τους φίλους μου

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες της διπλωματικής εργασίας κ. Γεώργιο Σταμούλη και κ. Νέστωρα Ευμορφόπουλο για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου με την ανάθεση του συγκεκριμένου θέματος, την άριστη συνεργασία και την συνεχή καθοδήγηση, η οποία διευκόλυνε την εκπόνηση της διπλωματικής εργασίας.

Επίσης θα ήθελα να ευχαριστήσω τους φίλους και συνεργάτες του εργαστηρίου Ε5 για την υποστήριξη και την δημιουργία ενός ευχάριστου και δημιουργικού κλίματος και ιδιαίτερα τον διδακτορικό φοιτητή Δημήτριο Γαρυφάλλου για τις εύστοχες υποδείξεις του και την συνεχή στήριξή του.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο κατά την διάρκεια των σπουδών μου όσο και κατά την εκπόνηση της διπλωματικής μου εργασίας.

Αναστάσιος Μελισσόπουλος

Βόλος, 2018

Contents

List of Figures

Περίληψη

Ακριβή μοντέλα βιβλιοθηκών που χρησιμοποιούνται για να περιγράψουν τη σύνθετη συμπεριφορά των τρανζίστορ των standard cells, ειδικά σε μικρές τεχνολογίες, μπορούν να παραχθούν με χαρακτηρισμό. Οι μηχανικοί υπολογιστών βασίζονται, σε μεγάλο βαθμό, σε αυτή την ακρίβεια ώστε να υλοποιήσουν ψηφιακή σχεδίαση και πρέπει να είναι απόλυτα σίγουροι ότι χρησιμοποιούν αξιόπιστες πηγές. Δυστυχώς οι βιβλιοθήκες ανοιχτού περιεχομένου δεν μπορούν να προσφέρουν πλήρη Composite Current Source (CCS) χαρακτηρισμό, η οποία είναι μια τεχνολογία σχεδιασμού μοντέλων που προσφέρουν μια πλήρη λύση βασισμένη στο ρεύμα για χρόνο, ισχύ και θόρυβο.

Το κίνητρο για αυτή την διπλωματική δόθηκε από την ανάγκη για να καλυφθεί το κενό στον CCS χαρακτηρισμό και μοντελοποίηση. Στις περισσότερες περιπτώσεις, με μια απλή παρατήρηση στις ανοιχτές βιβλιοθήκες μπορούμε να παρατηρήσουμε ότι η μοντελοποίηση εισόδων και εξάρτησης καταστάσεων παραλείπεται για τον CCS power, οδηγώντας έτσι σε ανακριβή μοντέλα.

Προσπαθούμε όχι απλά να κάνουμε έναν σωστό CCS χαρακτηρισμό αλλά και να δημιουργήσουμε ένα ολοκληρωμένο μοντέλο βιβλιοθήκης, σε δομή Liberty, το οποίο θα συμπεριλαμβάνει μη γραμμικό μοντέλο καθυστέρησης (NLDM), μη γραμμικό μοντέλο ισχύος (NLPM), CCS timing και CCS power. Για την υλοποίηση χρησιμοποιούμε το εργαλείο Siliconsmart, παρέχεται από την Synopsys, το οποίο παράγει ακριβή μοντέλα βιβλιοθηκών είτε από το μηδέν είτε ξαναχαρακτηρίζοντας ένα υπάρχων. Για να διαμορφώσουμε το εργαλείο και να πραγματοποιήσουμε όλη την διαδικασία χαρακτηρισμού χρησιμοποιούμε σκριπτάκια σε tcl.

Λέξεις κλειδιά:
Standard Cell, Χαρακτηρισμός, Composite Current Source, Liberty, Μη Γραμμικό Μοντέλο Καθυστέρησης, Μη Γραμμικό Μοντέλο Ισχύος, Siliconsmart, Synopsys, Tcl

Abstract

Accurate library models used to describe complex transistor behavior of cells, especially in small process nodes, can be produced by the process of characterization. Computer engineers rely, to a large extent, on this accuracy in order to implement their digital designs and must be totally sure that they use reliable sources. Unfortunately open libraries cannot deliver a completed Composite Current Source (CCS) characterization, a modeling technology which delivers a complete current based solution for timing, power and noise.

This thesis was motivated by the need for filling the gap on CCS power characterization and modeling. In most cases, by a simple observation in open libraries we can see that state dependency and input-only modeling is omitted for CCS power, leading to inaccurate models.

We try not only to make a correct CCS power characterization, but also to create a completed library model, in Liberty format, including Non Linear Delay Model (NLDM), Non Linear Power Model (NLPM), CCS timing and CCS power. For the implementation we use the Siliconsmart tool, provided by Synopsys, which generates accurate model libraries either from scratch or by recharacterizing an existing once. In order to configure the tool and run the characterization process we use tcl scripts.

Keywords:
Standard Cell, Characterization, Composite Current Source, Liberty, Non Linear Delay Model, Non Linear Power Model, Siliconsmart, Synopsys, Tcl script

Chapter 1

Introduction

1.1. Problem description

Designing at nanometer process technologies, especially at advanced nodes, requires high accuracy and therefore library models must accurately capture the complex transistor behavior of cells. Composite Current Source (CCS) is a state of the art industry gate level model which delivers a complete current-based solution for timing, noise and power. While there are many open-source standard cell libraries available which deliver a full CCS-timing and noise characterization, CCS-power is deficient and imprecise. Power analysis of integrated circuits based on these libraries is untrustworthy and the need for accuracy is increasingly rising.

1.2. Thesis contribution

Through a detailed characterization not only for CCS-power but also for NLDM, NLPM and CCS-timing, this thesis forms a complete guide that anyone can use to implement his own characterization including timing and power analysis. Our experiments were conducted on NanagateOpenCellLibrary PDK45 [1] and Globalfoundries 40nm low power (LP) [2].

For the implementation we used the Siliconsmart tool, provided by Synopsys, which is a comprehensive characterization solution for standard cells and supports for the latest modeling formats including all CCS models. SiliconSmart is ready for characterizing and modeling libraries at advanced technology nodes, such as 16-nm and 14-nm. Its built-in FineSim™ simulation technology and tight integration with the gold-standard HSPICE® circuit simulator enable characterization and signoff accuracy [3].

Starting from setting up Siliconsmart on our pc, we created and executed tcl scripts in order to call and give the suitable parameters to the tool. The results of characterization are stored in Liberty modeling format (.lib) and can be easily accessible.

1.3. Formation of the thesis

In chapter 2 we give background material on standard cell libraries and characterization. We also introduce Siliconsmart, explaining all available characterization flows and choosing the one that we are going to use in this thesis. In chapter 3 we give information about timing characterization analyzing NLDM and CCS-timing. Furthermore, in chapter 4 we give information about power characterization analyzing NLPM and CCS-power. We provide also in chapter 5 detailed information about how characterization is done so that anyone with basic characterization knowledge can run a flow. Finally in chapter 6 we make a conclusion and give some tips for further future improvements of our implementation.

Chapter 2

Characterization of Standard Cell Library

2.1. Standard cell

A Standard cell [4] is a group of transistor and interconnect structures that provides a Boolean logic function (e.g., AND, OR, XOR, NOR gate) or a storage function (flipflop or latch). The simplest cells are direct representations of the element NAND, NOR and XOR Boolean function, although cells of much greater complexity are commonly used (such as a 2-bit full-adder, or muxed D-input flipflop). The cell's boolean logic function is called its logical view: functional behavior is captured in the form of a truth table or Boolean algebra equation (for combinational logic), or a state transition table (for sequential logic).

2.2. Standard cell Library

A standard cell library is a collection of low-level electronic logic functions such as AND, OR, INVERT, flip-flops, latches, and buffers. These cells are realized as fixed-height, variable-width full-custom cells. The key aspect with these libraries is that they are of a fixed height, which enables them to be placed in rows, easing the process of automated digital layout. The cells are typically optimized full-custom layouts, which minimize delays and area.

A typical standard-cell library contains two main components:

- Library Database - Consists of a number of views often including layout, schematic, symbol, abstract, and other logical or simulation views. From this, various information may be captured in a number of formats including the Cadence LEF format, and the Synopsys Milkyway format, which contain reduced information about the cell layouts, sufficient for automated "Place and Route" tools.
- Timing Abstract - Generally in Liberty format, to provide functional definitions, timing, power, and noise information for each cell.

2.3. Library Characterization

Cell library characterization is a process of analyzing a circuit using static and dynamic methods to generate models suitable for chip implementation flows. It is needed because accurate library characterization is the cornerstone of successful digital implementation. Synthesis, place-and-route, verification and signoff tools rely on detailed model libraries to accurately represent the timing, noise and power performance of digital and memory designs. The complexity of cell libraries dramatically increases as designs migrate to smaller process nodes. Process variability on these nodes requires fast characterization on hundreds of corners. Furthermore, foundries are constantly updating SPICE models, requiring repeated characterization runs. Low-power SoC design further complicates the characterization process by introducing complex cells such as multi-bit flip-flops, multivoltage level shifters and retention logic, which must be accurately characterized to ensure effective digital implementation across multiple power domains.

2.4. Software Tool for Characterization

The Siliconsmart tool [3] is used for the process of characterization in this thesis. It is a comprehensive characterization solution for standard cells, I/O, complex cells and memory. It generates accurate model libraries tightly correlated with Synopsys' digital implementation tools. Its built-in FineSim simulation technology and tight integration with the gold-standard HSPICE® circuit simulator enable characterization and signoff accuracy. SiliconSmart supports all of the standard models, including NLDM (non-linear delay model), CCS (composite current source) and AOCV (advanced on-chip variation) models.

2.5. Characterization Flows

A critical point is the selection of the characterization flow. There are two sections to determine it as described below [5]:

2.5.1. Selection by starting point

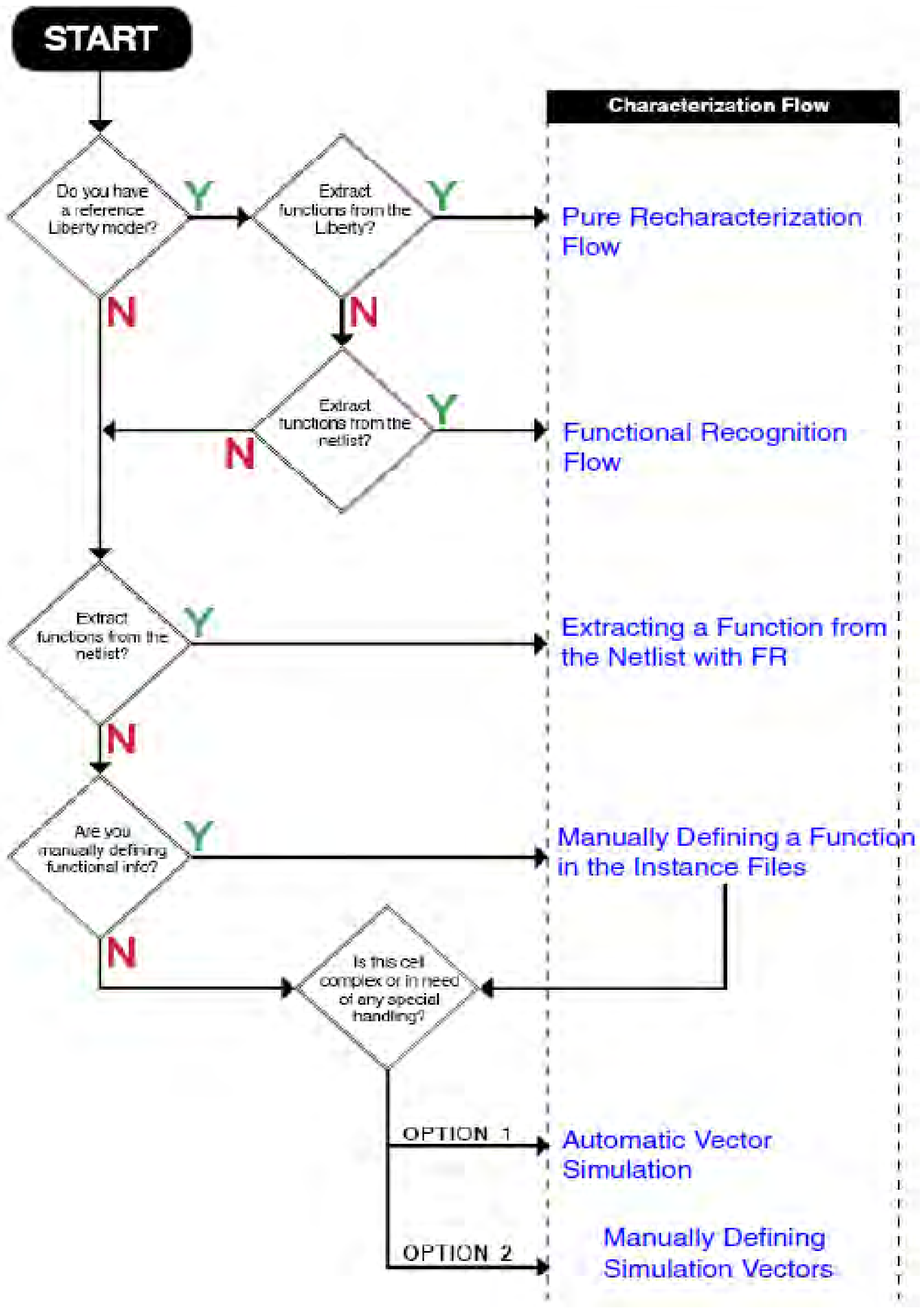

START
Characterization Flow
Do you have a reference Liberty model?
Y
N
Extract functions from the Liberty?
Y
N
Pure Recharacterization Flow
Extract functions from the netlist?
Y
N
Functional Recognition Flow
Extract functions from the netlist?
Y
N
Extracting a Function from the Netlist with FR
Are you manually defining functional info?
Y
N
Manually Defining a Function in the Instance Files
Is this cell complex or in need of any special handling?
OPTION 1
Automatic Vector Simulation
OPTION 2
Manually Defining Simulation Vectors

Figure 2.1 Selecting Characterization Flow

As we can notice in figure 2.1 that there are six different paths depending on the Liberty model availability and the function extraction from the netlist or the Liberty.

2.5.2. Selection by characterization approach

The basic approaches for characterizing with the Siliconsmart tool are:

- **Recharacterization Flow** -- This approach will recharacterize data in an existing Liberty model or add new data to an existing Liberty model.
  - **Pure Recharacterization Flow** -- this flow extracts all necessary data from an existing Liberty model. It requires minimal input from the user.
  - **Functional Recognition Flow** -- this specialized flow recognizes the function from the netlist of a cell and extracts the slews/loads/timing arcs from an existing Liberty model, removing the dependency on the function attributes in a Liberty model.
  - **Skeleton Liberty-Based Flow** -- this flow generates a new Liberty model from scratch (employing user inputs for load/slope/when conditions) while preserving the attributes and other groups from the reference Liberty. Any processing performed previously on the reference Liberty can be carried over into the brand new Liberty model.
  - **Incremental Characterization Flow** -- this is another specialized form of the Recharacterization flow which allows the user to only characterize a new view and adds to an existing Liberty model while preserving all the original data as is. An example would be to add CCS-Noise to an existing NLDM Liberty.
- **Function-Based Flow** -- The function-based approach uses functions to automatically determine configuration.
- **Structure-Based Flow** -- A function-based approach that uses functions and automatic vector simulation based on the structure of the circuit.
- **Sequence-Based Flow** -- The sequence based approach adds a user defined input stimulus to specify arcs to be characterized and the sequence to be performed, without using automated methods.

In this thesis we work with **skeleton liberty-based flow.** This flow maintains only the attributes and structure of the Liberty while using recharacterization to create a new Liberty model. This is primarily useful in flows where the user can provide custom load/slope/when conditions but wants to keep the attributes, custom library modifications, headers, comments, etc., from the imported Liberty. This flow uses the *-skeleton* switch, available for the *import* and *model* commands, which does the following:

- The SiliconSmart tool will discard all timing, constraint*, internal_power*, and CCS timing/power/noise tables from the reference Liberty.
- As usual, an instance file will be created in the $charpoint/control directory for each cell with the interface and functional information. All of the load/ slope/when conditions from the reference Liberty will be discarded.

Chapter 3

Timing Models and Characterization

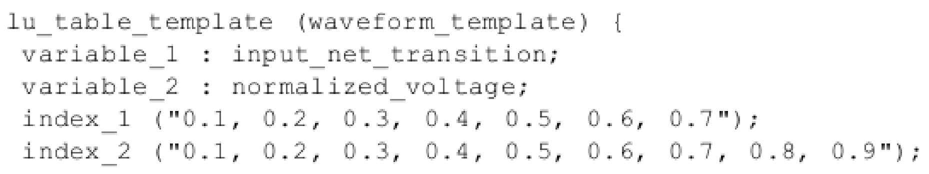
3.1. Timing Models

The techniques that Siliconsmart uses, and we are interested in, to capture the timing characteristics of standard cells in a library are [5]:

- **Timing Measurements**
- **Constraints**

3.1.1. Timing Measurements

The most basic timing measurements are the **propagation delay** through a cell from an input to an output and the output signal **transition time**. For basic combinational cells, this means applying a range of input transitions to the cells over a range of output loads and measuring the delay and output transition times. The results are two-dimensional tables, as we can see in figure 3.1, of delays and transition times indexed by input transition time and output load [6].



```
lu_table_template (waveform_template) {
 variable_1 : input_net_transition;
 variable_2 : normalized_voltage;
 index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
 index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
```

Figure 3.1 Lu_table_template Structure

Propagation delays are measured from the time when the input transition crosses the logic delay threshold to when the output transition crosses the same threshold. The threshold is specified as a fraction of the voltage swing of the pin by setting the pin type parameter *logic_low_threshold* to 50% (default value) in the Pin Type Definitions block of the configure file.

Similarly the output transition time is measured from the time when the output signal crosses the low (high) threshold to when the signal crosses the high (low) threshold for rising (falling) transitions. The thresholds are specified by setting the pin type parameters *logic_low_threshold* to 30% and *logic_high_threshold* to 70% in the Pin Type Definitions block of the configure file. In figure 3.2 we can see how they are modeled as a part of the cell library.

```
/* threshold definitions */
slew_lower_threshold_pct_fall  : 30.0;
slew_upper_threshold_pct_fall  : 70.0;
slew_lower_threshold_pct_rise  : 30.0;
slew_upper_threshold_pct_rise  : 70.0;
input_threshold_pct_fall       : 50.0;
input_threshold_pct_rise       : 50.0;
output_threshold_pct_fall      : 50.0;
output_threshold_pct_rise      : 50.0;
```

Figure 3.2 Threshold Definitions Model

A timing arc is positive unate if a rising transition on an input causes the output to rise or not to change and a falling transition on an input causes the output to fall or not to change as we see in figure 3.3. A negative unate timing arc occurs when a rising transition on an input causes the output to have a falling transition and a falling transition on an input causes the output to have a rising transition or no change. A non-unate timing arc occurs when the output transition cannot be determined only from the direction of input but will also depend on the state of the side inputs.

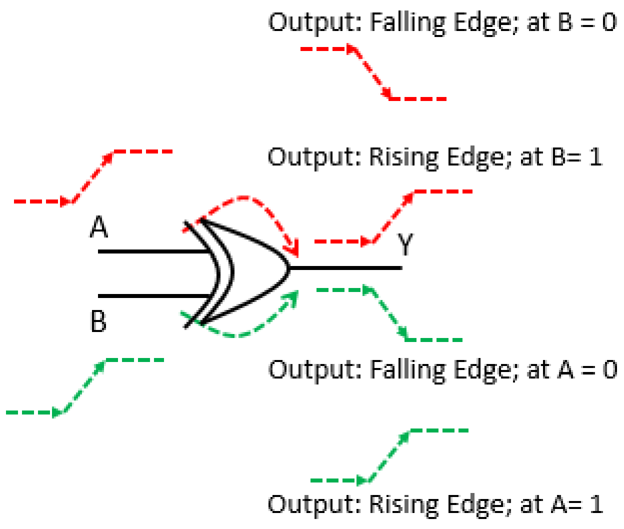

Output: Falling Edge; at B = 0
Output: Rising Edge; at B= 1
A
Y
B
Output: Falling Edge; at A = 0
Output: Rising Edge; at A= 1

Figure 3.3 Timing Arc

An other group of timing measurements is **current course models** which describes transitions as a waveform instead of as a single transition time. The format that we choose to implement is CCS, which is supported by Synopsys, and it stores current waveforms, originated from converted voltage waveforms, for each point in the transition tables. CCS waveforms record, specifically, the current through the load capacitor at a set of time points during the transition and the data is stored in the standard Liberty units as we can see in figure 3.4 for *output_current_rise* block for different reference times. The *reference_time* attribute represents the time at which the input waveform crosses the rising of falling input delay threshold. *Index_1* refers to input transition, *index_2* to output capacitance and *index_3* to time.

```
timing() {
  output_current_rise () {
    vector (template_name_id){
      reference_time : float;
      index_1 (float);
      index_2 (float);
      index_3 ("float,..., float");
      values("float,..., float");
    }
  }
```

Figure 3.4 output_current Model

The last timing measurements, that we are interested in, are **timing receiver models** which can be divided to **arc-level** where models are modeled inside a timing() group and can be modeled for all input pins from where a timing arc originates and **pin-level** where models are modeled inside a pin() group and typically modeled for all those input pins from where no direct timing arc originates.

Timing modeling with CCS is composed of a driver model and a receiver model. The CCS timing receiver model uses two capacitance values to model the variation of cell input capacitance during the input signal transition. In conventional two-segment receiver capacitance models, which we are going to use in this thesis, the voltage rise (or fall) at in input or inout is divided into two segments and the corresponding capacitance values are stored in the *receiver_capacitance1_rise/ reciver_capacitance1_fall* and *receiver_capacitance2_rise/ reciver_capacitance2_fall* groups as we can see in figure 3.5 for an output pin.

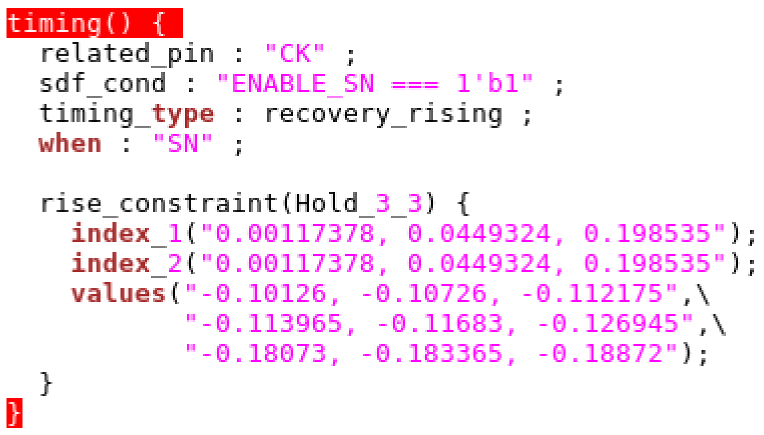
```
  pin(pin_name) {
    direction : output; /* or "inout" */
    timing() {
      when : "Boolean_expression";
      mode (mode_name, mode_value);
      ...
      receiver_capacitance() {
        receiver_capacitance1_rise (lu_template_name) {
          index_1("float, ..., float");
          index_2("float, ..., float");
          values("float, ..., float");
        }
        receiver_capacitance1_fall (lu_template_name)
{ ... }
        receiver_capacitance2_rise (lu_template_name)
{ ... }
        receiver_capacitance2_fall (lu_template_name)
{ ... }
      }
    } ...
  }
```

Figure 3.5 receiver_capacitance Model

3.1.2. Constraints

Timing constraints examine the relative timing between two input transitions. The transitions may occur on two different pins, such as in setup and hold measurements, or on a single input as in minimum pulse width measurements. In all cases the measurement seeks to find the minimum spacing that can occur between the two edges before the cell fails to operate as expected.

As we do not focus on these measurements in this thesis, default constraints measurements are quite sufficient for us. As a matter of fact, setup, hold, removal and recovery falling/rising measurements are held in the library. In figure 3.6 we can see an example of recovery rising case.



```
timing() {
  related_pin : "CK" ;
  sdf_cond : "ENABLE_SN === 1'b1" ;
  timing_type : recovery_rising ;
  when : "SN" ;

  rise_constraint(Hold_3_3) {
    index_1("0.00117378, 0.0449324, 0.198535");
    index_2("0.00117378, 0.0449324, 0.198535");
    values("-0.10126, -0.10726, -0.112175",\
           "-0.113965, -0.11683, -0.126945",\
           "-0.18073, -0.183365, -0.18872");
  }
}
```

Figure 3.6 Constraint Measurements

3.2. CCS Timing Characterization

CCS Timing consists of a driver model and a receiver model. The driver model describes how a timing arc propagates a transition from input to output, and how it can drive arbitrary RC networks. The receiver model describes the capacitance that an input pin presents to driving cells [7]

CCS Timing delay calculation uses advanced interpolation technology to determine a current waveform when the input slew and/or output load values do not match those used during cell characterization. Additionally, interpolation is used for intermediate values of VDD and temperature by using data from multiple libraries.

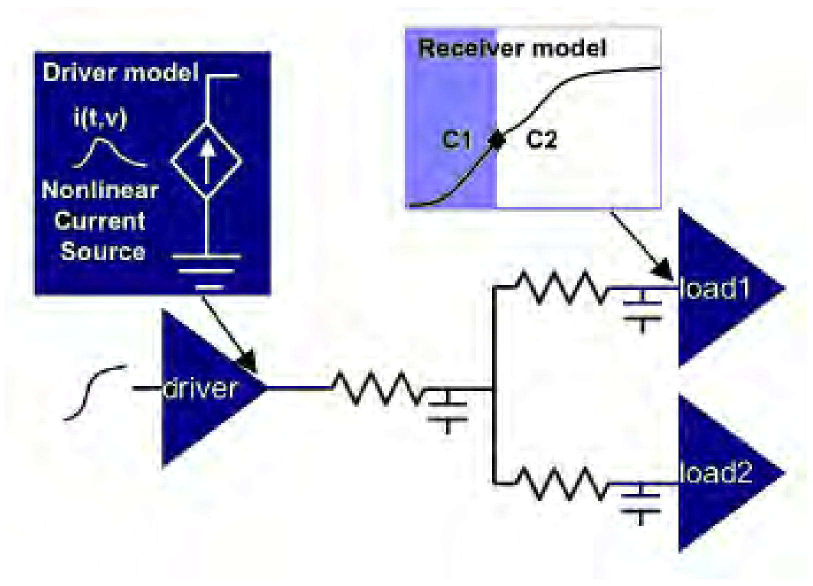


Figure 3.7 Timing Model

Characterizing a cell timing arc for CCS Timing is very similar to characterization for nonlinear delay models (NLDM): an input stimulus is chosen to produce a specific input slew time (S_{inp}); a load capacitance (C_{out}) is connected to the output pin; and a circuit simulation is run in the same way as for NLDM. But instead of measuring voltage thresholds at the output pin, current is measured through the load capacitor and into the input pin. The current through C_{out} is used for the driver model, and the current into the input pin is used to determine the receiver model.

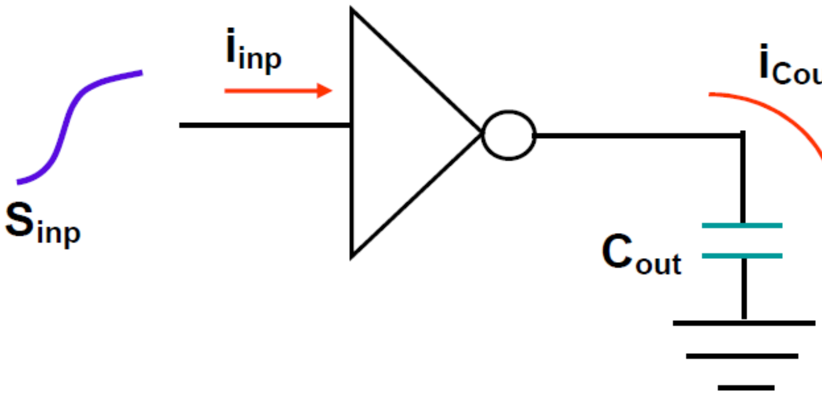


Figure 3.8 CCS Timing Characterization Measurements

These characterization experiments are repeated for a table of different S_{inp} and C_{out} combinations. The current through C_{out} is saved for every circuit simulation timestep and then reduced to a much smaller set of current and time (i, t) points. The points are chosen such that $V_{out}(t)$ can be accurately reproduced for every timestep during the transition. Figure 3.9 shows an example of the complete i(t) waveform and a reduced set of points.

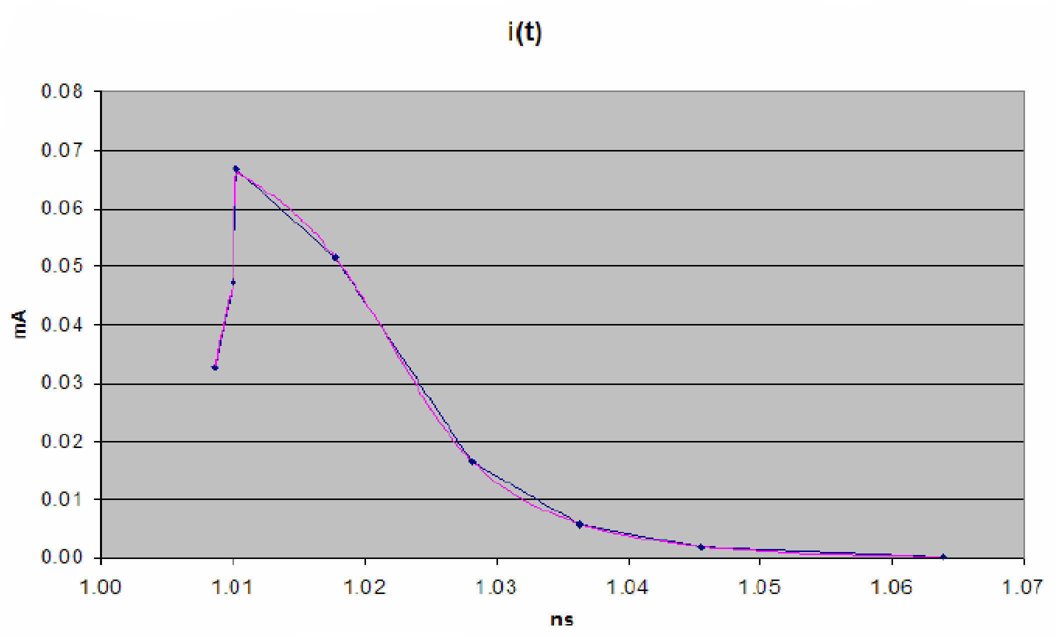


Figure 3.9 Current Waveform from Circuit Simulation and Reduced Current Points

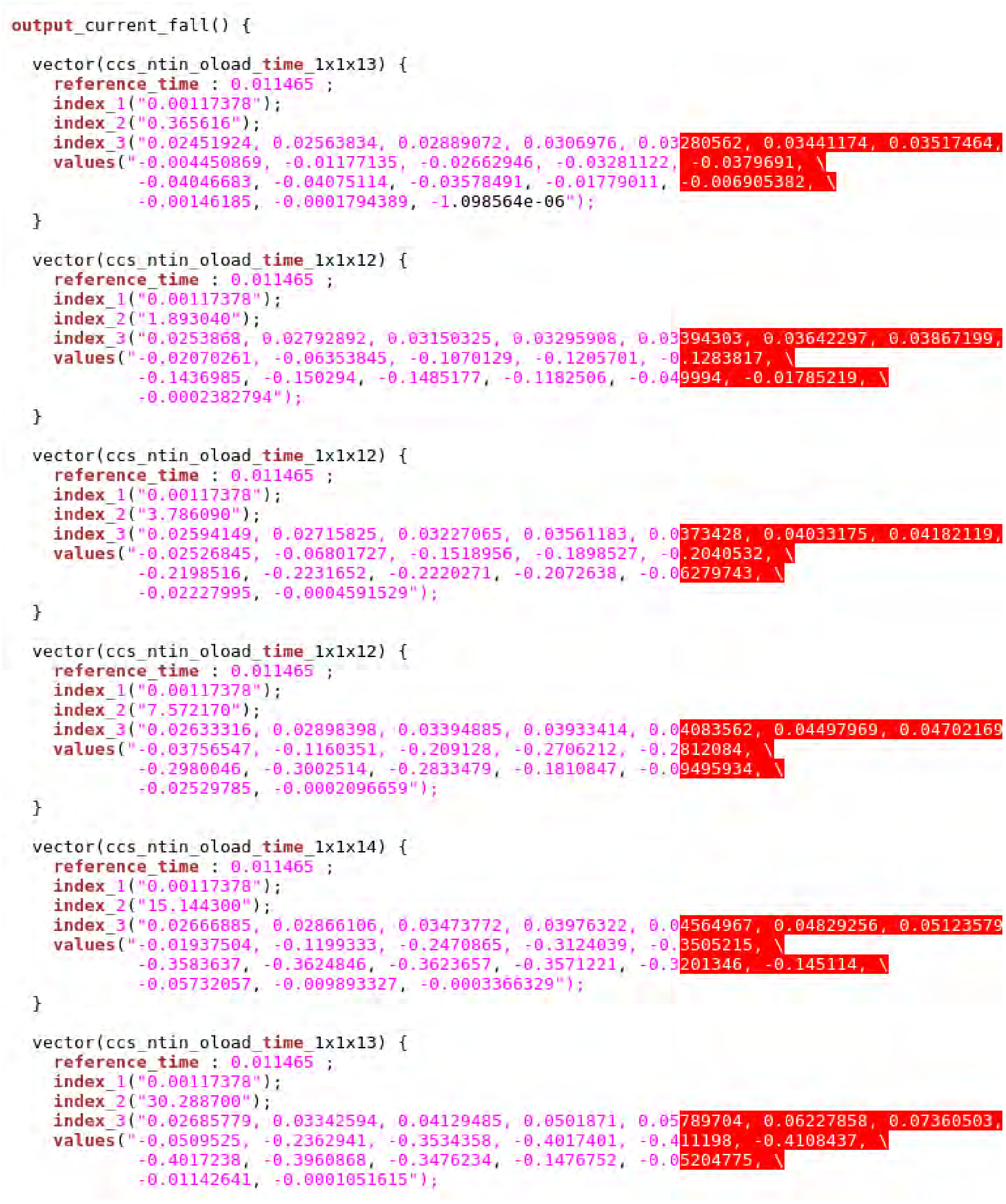
The current and voltage at the input pin are saved and then used to determine C1 and C2 values such that gate-level delay calculation can closely match times to the delay threshold and to the second slew threshold at the input pin.

An additional piece of information, input reference time, is needed in order to calculate cell delays. The reference time is the simulation time at which the waveform at the input pin crosses the rising or falling delay threshold (often this is 50% of VDD).

Benefits of CCS Timing:

- The CCS Timing delay calculation provides a high accuracy response for cell delay, interconnect delay, and pin slew.
- The CCS Timing receiver model produces excellent results on single-stage cells with large Miller effect. Furthermore, the stage delay and slew results are typically within 2% of the golden circuit simulation values.
- The CCS Timing enables scaling for intermediate VDD and temperature values. Synopsys delay calculation with CCS Timing includes powerful nonlinear Vdd scaling for timing check arcs. This results in better correlation to circuit simulation than with simple linear interpolation approaches.
- The current waveforms are expected to consume larger space in terms of data size compared to the NLDM models. Therefore, a "Compact CCS" is used to represent the current waveforms in a very compact form. The compact CCS takes advantage of similarity of I/V curves in the library. The compact CCS modeling uses a common set of I/V curves (known as base-curves) for the entire library and each instantiation of the current waveform is derived from one of these base curves. This technique allows for high accuracy while reducing the library size by up to 3 to 4x compared to the expanded (non-compact) CCS timing library.
- CCS timing also allows, for accurate representation of current characteristics of the library subjected to the process, variation. The variation-aware extension of CCS timing captures the current waveforms as the cell is subjected to process variation with respect to the process parameters.

In figure 3.10 we can see an example of an output_current section for a CCS timing characterization.



```
output_current_fall() {

  vector(ccs_ntin_oload_time_1x1x13) {
    reference_time : 0.011465 ;
    index_1("0.00117378");
    index_2("0.365616");
    index_3("0.02451924, 0.02563834, 0.02889072, 0.0306976, 0.03280562, 0.03441174, 0.03517464,
    values("-0.004450869, -0.01177135, -0.02662946, -0.03281122, -0.0379691, \
            -0.04046683, -0.04075114, -0.03578491, -0.01779011, -0.006905382, \
            -0.00146185, -0.0001794389, -1.098564e-06");
  }

  vector(ccs_ntin_oload_time_1x1x12) {
    reference_time : 0.011465 ;
    index_1("0.00117378");
    index_2("1.893040");
    index_3("0.0253868, 0.02792892, 0.03150325, 0.03295908, 0.03394303, 0.03642297, 0.03867199,
    values("-0.02070261, -0.06353845, -0.1070129, -0.1205701, -0.1283817, \
            -0.1436985, -0.150294, -0.1485177, -0.1182506, -0.049994, -0.01785219, \
            -0.0002382794");
  }

  vector(ccs_ntin_oload_time_1x1x12) {
    reference_time : 0.011465 ;
    index_1("0.00117378");
    index_2("3.786090");
    index_3("0.02594149, 0.02715825, 0.03227065, 0.03561183, 0.0373428, 0.04033175, 0.04182119,
    values("-0.02526845, -0.06801727, -0.1518956, -0.1898527, -0.2040532, \
            -0.2198516, -0.2231652, -0.2220271, -0.2072638, -0.06279743, \
            -0.02227995, -0.0004591529");
  }

  vector(ccs_ntin_oload_time_1x1x12) {
    reference_time : 0.011465 ;
    index_1("0.00117378");
    index_2("7.572170");
    index_3("0.02633316, 0.02898398, 0.03394885, 0.03933414, 0.04083562, 0.04497969, 0.04702169
    values("-0.03756547, -0.1160351, -0.209128, -0.2706212, -0.2812084, \
            -0.2980046, -0.3002514, -0.2833479, -0.1810847, -0.09495934, \
            -0.02529785, -0.0002096659");
  }

  vector(ccs_ntin_oload_time_1x1x14) {
    reference_time : 0.011465 ;
    index_1("0.00117378");
    index_2("15.144300");
    index_3("0.02666885, 0.02866106, 0.03473772, 0.03976322, 0.04564967, 0.04829256, 0.05123579
    values("-0.01937504, -0.1199333, -0.2470865, -0.3124039, -0.3505215, \
            -0.3583637, -0.3624846, -0.3623657, -0.3571221, -0.3201346, -0.145114, \
            -0.05732057, -0.009893327, -0.0003366329");
  }

  vector(ccs_ntin_oload_time_1x1x13) {
    reference_time : 0.011465 ;
    index_1("0.00117378");
    index_2("30.288700");
    index_3("0.02685779, 0.03342594, 0.04129485, 0.0501871, 0.05789704, 0.06227858, 0.07360503,
    values("-0.0509525, -0.2362941, -0.3534358, -0.4017401, -0.411198, -0.4108437, \
            -0.4017238, -0.3960868, -0.3476234, -0.1476752, -0.05204775, \
            -0.01142641, -0.0001051615");
```

Figure 3.10 Example of output_current Section

Chapter 4

Power Models and Characterization

4.1. Power Models

Siliconsmart uses the following library characterization for power [8]:

- Non Linear Power Model (NLPM) format based on the voltage-source circuit elements.
- Composite Current Source (CCS) Power format based on the current-source circuit elements.

The NLPM power model is an abstract power model for a cell with leakage power information and internal power information in the form of internal energy data for power analysis.

The CCS power model is an advanced current-based modeling technology data on the PG pins with the leakage current for leakage power and instantaneous power data in the form of current waveform data form for internal power. It includes also the gate leakage information. The following are features of this approach as compared to the nonlinear power model:

- Creates a single unified power library format suitable for power optimization, power analysis and rail analysis.
- Captures a supply current waveform for each power or ground pin
- Provides finer time resolution.
- Offers full multivoltage support.
- Captures equivalent parasitic data to perform fast and accurate rail analysis.
- Reduces the characterization time.

The power consumption of a standard cell is categorized into the following:

- static or leakage power
- dynamic power

4.1.1. Static or Leakage Power

Leakage power is the power consumed by a cell when it is in a steady-state condition. The liberty syntax [6] represents leakage power information in the library as:

- Library-level leakage power with the *default_cell_leakage_power* attribute for NLPM.

- Cell-level state-independent leakage power with the *cell_leakage_power* attribute for NLPM. If this attribute is missing or negative, the value of the *default_cell_leakage_power* attribute is used.
- Cell-level state-dependent leakage power with the *leakage_power* group for NLPM. *Related_pg_pin* attribute is used to associate a power and ground pin with leakage power, *when* specifies the state-dependent condition that determines whether the leakage power is accessed and *value* represents the leakage power for the given state.
- Cell-level state-dependent leakage current with the *leakage_current* attribute for CCS power. *Pg_current* group specifies a power or ground pin where leakage current is to be measured.
- Cell-level state-dependent leakage current with the gate_leakage attribute for CCS power. This group specifies the cell's gate leakage current on input or inout pins. *Input_low_value* attribute specifies gate leakage current when the pin is in a *low* state and *input_high_value* when is in a *high* sate.
- Cell-level state state-dependent intrinsic parasitic model with *intrinsic_parasitic* attribute for CCS power. The *mode* attribute pertains to a individual cell. The cell is active when the mode attribute is instantiated with a name and a value.
- The associated library-level attributes that specify scaling factors, units and a default for both leakage and power density.

Syntax for NLPM:

```
library (my_lib_nlpm) {
…

default_cell_leakage_power : float;

..
cell (cell_name) {
...
   leakage_power () {
         related_pg_pin : pg_pin1;



      when : " boolean expression ";
      value : float;
}
cell_leakage_power : float;
…
```

Figure 4.1: NLPM Modeling for Leakage Power

Syntax for CCS power:

```
cell(cell_name) {
...
    leakage_current() {
      when : "boolean expression";
      pg_current(pg_pin_name) {
      value : float;
    }
    ...
    leakage_current() { /* without the when statement */
    /* default state */
    ...
    }
    gate_leakage(input pin name) {
      input_low_value : float;
      input_high_value : float;
    }
...
}
```

Figure 4.2: CCS Power Modeling for Leakage Power

```
cell (cell_name) {
  mode_definition (mode_name) {
    mode_value(namestring) {
    when : "Boolean expression";
    sdf_cond : "Boolean expression";
  }
  ...
  intrinsic_parasitic() {
    mode (mode_name, mode_value);
    when : "Boolean expression";
    intrinsic_resistance(pg_pin_name) {
      related_output : output_pin_name;
      value : float;
    }
    intrinsic_capacitance(pg_pin_name) {
      value : float;
    }
  }

  intrinsic_parasitic() {
    /*without when statement */
    /* default state */

  }
}
```

Figure 4.3: Intrinsic Parasitic Model for CCS Power

4.1.2. Dynamic Power

Dynamic power is the power consumed by a cell when it is active. It includes :

- internal power
- switching power

Internal power is the power dissipated by the charging or discharging of any existing capacitances internal to the cell. The liberty syntax represents internal power information in the library as:

- Library-level *power_lut_template* group for NLPM. It creates a template of the index used by the *internal*_power group.
- Pin-level internal energy tables with the *internal_power* group for NLPM. *Rise_power* group is accessed when the pin has a rise transition. If we have a *rise_power*, we must have a *fall_power* as well.
- Library-level *pg_current_template* group for CCS power.
- Pin-level current waveform tables with the *dynamic_current* group for CCS power. *Switching_group* group is used to specify a current waveform vector when the power and ground current is dependent on pin switching conditions. The *input_switching_condition* attribute specifies the sense of the toggling input and its valid values are *rise* and *fall.rise* represents a rising pin and *fall* represents a falling pin. Same apply for *output_switching_condition* but for output. Furthermore the *pg_current* group specifies current waveform data in a vector group. This group represents a single current waveform based on specified input slew and output load. The index attributes specify values for variables specified in the *pg_current_template.*
- The associated library-level attributes that specify the scaling factors and a default.

The internal power tables use the following indexes:

- The related input transition time for the non-propagating input pins.
- The related input transition time and the output capacitive load for the propagating output pins.

Syntax for NLPM:

```
 cell (cell_name) {
 ...

   pin ( pin_name) {
     …
     internal_power() {
     related_pin : " pin_name "
     related_pg_pin : "pg_pin1";
     rise_power( power_template_name ) {
            ..
            );
     }
     fall_power( power_template_name ) {
            …
            );
     }
   }/* end of internal power */

}
```

Figure 4.4: NLPM Model for Internal Power

Syntax for CCS power:

```
 pg_current_template(template_name_1) {
   variable_1 : input_net_transition;
   variable_2 : total_output_net_capacitance;
   variable_3 : time;
   index_1(float, );                  /* optional */
   index_2(float, );                  /* optional */
   index_3(float, );                  /* optional */
 }
```

Figure 4.5: CCS Power pg_current Template

```
cell(cell_name) {

  power_cell_type : enum(stdcell, macro)
  dynamic_current() {
      when : "Boolean expression";
      related_inputs : input_pin_name;
      related_outputs : output_pin_name;
      typical_capacitances(float, );/* applied for cross type;/*

      switching_group() {
           input_switching_condition(enum(rise, fall));

           output_switching_condition(enum(rise,
fall));
           pg current(pg pin name) {
                vector(template_name) {
                     reference_time : float;
                        index_output : output_pin_name
; /* applied for
cross type;/*
                     index_1(float);

                     index_n(float);
                     index_n+1(float, );
                     values(float, );
                }       /* vector */

            }      /pg_current */

        }  /* switching_group */

  } /* dynamic_current */

} /* cell */
```

Figure 4.6: CCS Power Model for Internal Power

Switching power, or capacitance power, of a driving cell is the power dissipated by the charging and discharging of the load capacitance at the output of a cell. The Liberty power models store only internal power component of the dynamic power. The switching power component is calculated during power analysis using the output switching activity and capacitive load.

4.2. Power Characterization

The CCS Power characterization process is very similar to NLPM characterization [9]:

1. First, the leakage currents are measured with simple DC analysis.
2. Next, the dynamic current waveforms are acquired with a transient analysis.
3. Finally, the equivalent parasitics is measured with fast AC and DC analysis runs.

We can perform most of the characterization for timing and power simultaneously to reduce the characterization runtime.

4.2.1. Leakage Power and Current

The leakage current and leakage power are related:

$$P_{leak} = I_{leak}*V_{supply} \tag{4.1}$$

For multi-supply pins we consider general case where there are multiple PG pins. The current of each pin is defined as the current that flows into the cell. For ground pins the current is negative. The total current for all pins must sum up to zero (current conservation). The power of a pin is defined as the product of the pin's current and its voltage relative to the reference node. The total leakage power of the cell is the sum of the leakage power of each pin:

$$P_{leak}\,(pg\_pin) = I_{leak}\,(pg\ pin) * V_{supply}\,(pg\_pin) \tag{4.2}$$

$$P_{leak} = \sum P_{leak}(pg\_pin) \tag{4.3}$$

The reference voltage impacts the power of the individual pins, but its value does not matter for the total leakage power because of current conservation.

The typical simulation setup for leakage characterization is shown in figure 4.7. This measurement is usually performed as part of the simulation setup for timing analysis

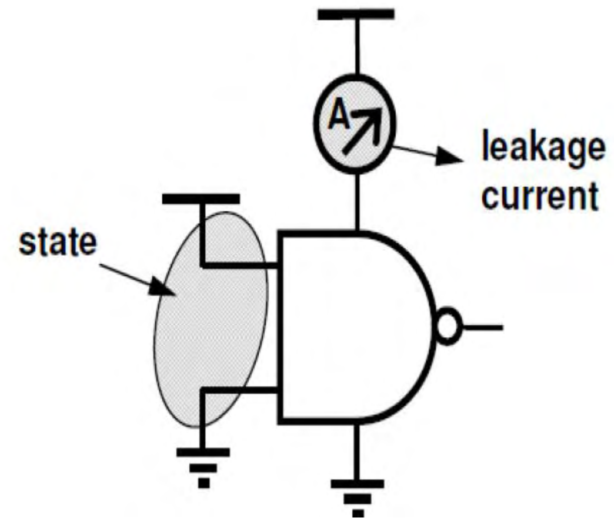


Figure 4.7: Leakage Current Measurement

While **gate leakage current** is quite insignificant in 100nm and higher technologies, it will gradually become a more significant current component in more advanced technology nodes.

Gate leakage is defined as a static current from a driver cell to transistor gates in a load cell. If the gate is driven high, the current flows from the power pin of the driver cell by way of the gate to the ground pin of the load cell. The current can also flow back from the power pin on the load cell via the gate to the ground pin of the driver cell. During the characterization process the leakage currents on the input and power/ground pins are measured for the cell in an open configuration, that is, with outputs that do not drive other cells. It is important that the cells' outputs do not conduct static current during the measurement.

The leakage currents are state dependent. The leakage current measurement is very similar to the measurement for the conventional model with only channel leakage: for combinational cells the inputs are driven at certain levels depending on the state, while for sequential cells a sequence of states may be required to bring the cell to a certain state. Once the transition is complete, the currents on the power/ground pins and input pins are measured.

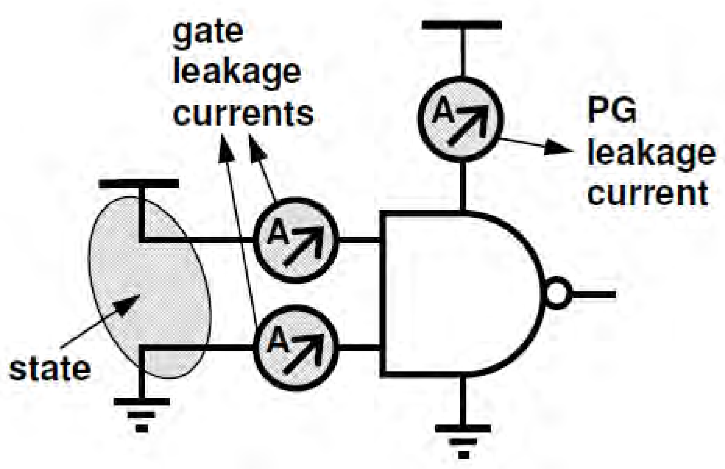


Figure 4.8 Gate Leakage Measurement

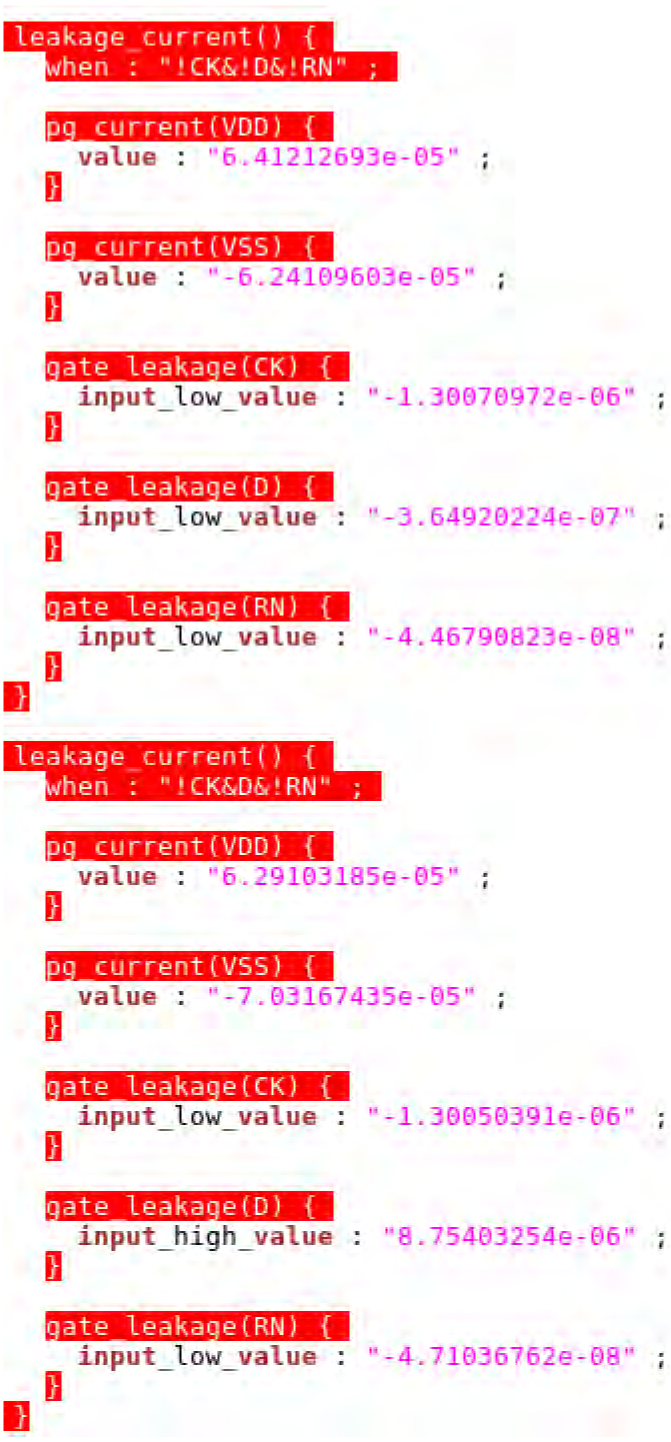
The direction of the current depends on the state of the input pin that is connected to the transistor gates. The magnitude of the current depends on the state of the other input pins of the load cell. During leakage power measurement you can measure the gate leakage currents on the input pins together with the channel leakage currents on the PG pins, as shown in figure 4.8. It is important to note that this measurement should be performed with open output pins so that no current is drawn from the outputs.

Current conservation still holds, but now the gate leakage of the input pins must be taken into account as well:

$$\sum I_{leak}(pg\_pin) + \sum I_{leak}(input\_pin) = 0 \qquad (4.4)$$

When a cell is present in a netlist, the output currents will not be zero. Instead, a certain constant current will be supplied to maintain the gate leakage of the driven cell. The application adds the gate leakage current of the driven cell to the power/ground pins of the driving cell to calculate the actual leakage currents in operation.

In figure 4.9 we can see an example of leakage current measurement for two states of a Dflipflop cell.


```
leakage_current() {
  when : "!CK&!D&!RN" ;

  pg_current(VDD) {
    value : "6.41212693e-05" ;
  }

  pg_current(VSS) {
    value : "-6.24109603e-05" ;
  }

  gate_leakage(CK) {
    input_low_value : "-1.30070972e-06" ;
  }

  gate_leakage(D) {
    input_low_value : "-3.64920224e-07" ;
  }

  gate_leakage(RN) {
    input_low_value : "-4.46790823e-08" ;
  }
}
leakage_current() {
  when : "!CK&D&!RN" ;

  pg_current(VDD) {
    value : "6.29103185e-05" ;
  }

  pg_current(VSS) {
    value : "-7.03167435e-05" ;
  }

  gate_leakage(CK) {
    input_low_value : "-1.30050391e-06" ;
  }

  gate_leakage(D) {
    input_high_value : "8.75403254e-06" ;
  }

  gate_leakage(RN) {
    input_low_value : "-4.71036762e-08" ;
  }
}
```
Figure 4.9 Example of leakage_current Section

4.2.2. CCS Decoupling Capacitance

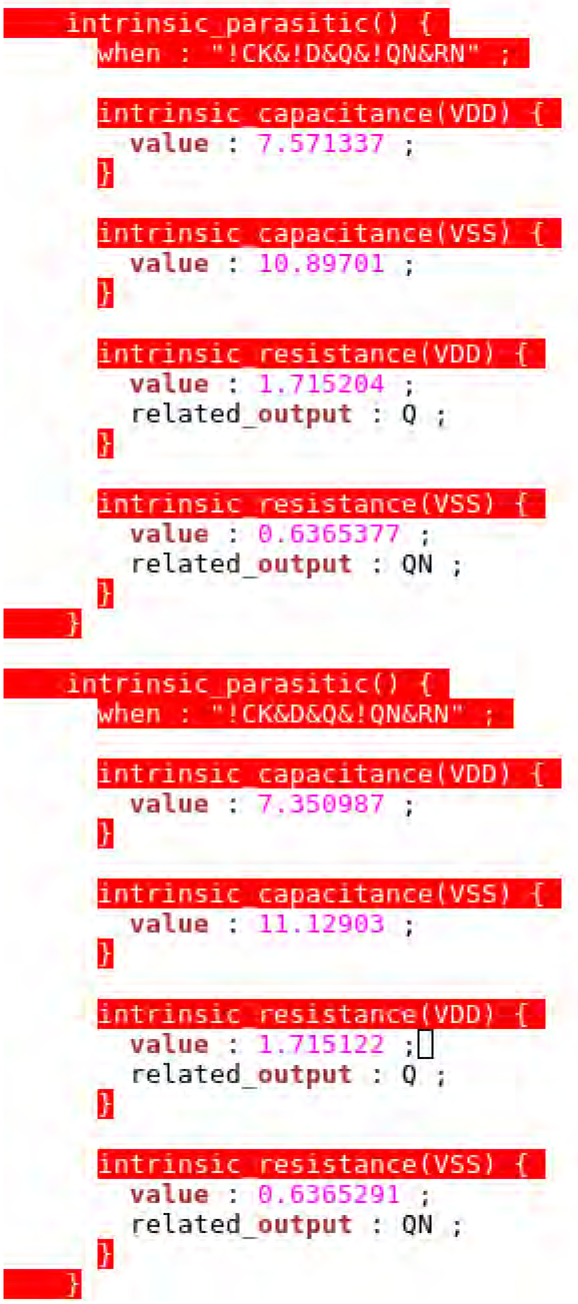
To characterize intrinsic parasitics [5], a small sinusoidal voltage perturbation is applied to each PG pin separately. All outputs are connected with tiny loading capacitance vales (that is, 1fF) such that the measured intrinsic capacitance is purely PG pin capacitance, and it does not depend on loading. The recommended voltage magnitude is 0.1V or 10% of power supply.

By measuring the magnitude and phase of the current response, the capacitance is calculated by the following formula:

$$\frac{I_0}{\omega V_0 \cos(\phi)}$$

where:
ω is the sinusoidal waveform frequency,
V_0 is the magnitude of the perturbation voltage waveform,
I_0 is the magnitude of the current response and
ϕ is the phase of the current response.

In figure 4.10 we can see how intrinsic parasitic modeled for two states of a Dflipflop cell.



```
    intrinsic_parasitic() {
      when : "!CK&!D&Q&!QN&RN" ;

      intrinsic_capacitance(VDD) {
        value : 7.571337 ;
      }

      intrinsic_capacitance(VSS) {
        value : 10.89701 ;
      }

      intrinsic_resistance(VDD) {
        value : 1.715204 ;
        related_output : Q ;
      }

      intrinsic_resistance(VSS) {
        value : 0.6365377 ;
        related_output : QN ;
      }
    }

    intrinsic_parasitic() {
      when : "!CK&D&Q&!QN&RN" ;

      intrinsic_capacitance(VDD) {
        value : 7.350987 ;
      }

      intrinsic_capacitance(VSS) {
        value : 11.12903 ;
      }

      intrinsic_resistance(VDD) {
        value : 1.715122 ;□
        related_output : Q ;
      }

      intrinsic_resistance(VSS) {
        value : 0.6365291 ;
        related_output : QN ;
      }
    }
```

Figure 4.10 Example for Intrinsic Parasitic Model

4.2.3. Dynamic and Internal Current

Dynamic energy that occurs during a transition is derived from the total charge associated with that transition and the voltage over which charge was transferred:

$$E_{dyn} = Q_{dyn} * V_{supply} \tag{4.4}$$

The charge is the integral of the dynamic current:

$$Q_{dyn} = \int_0^{+\infty} I_{dyn}(t)dt \tag{4.5}$$

Similar to internal energy, we can derive the internal current from the dynamic current by subtracting the switching component. The switching current corresponds to the output current that drives the load that is being (dis)charged.

$$I_{int} = I_{dyn} - I_{sw} \tag{4.6}$$

The dynamic current waveforms are acquired by performing a transient analysis as shown in figure 4.11. This setup is identical to that used for timing characterization and therefore we can perform timing and power characterization simultaneously:

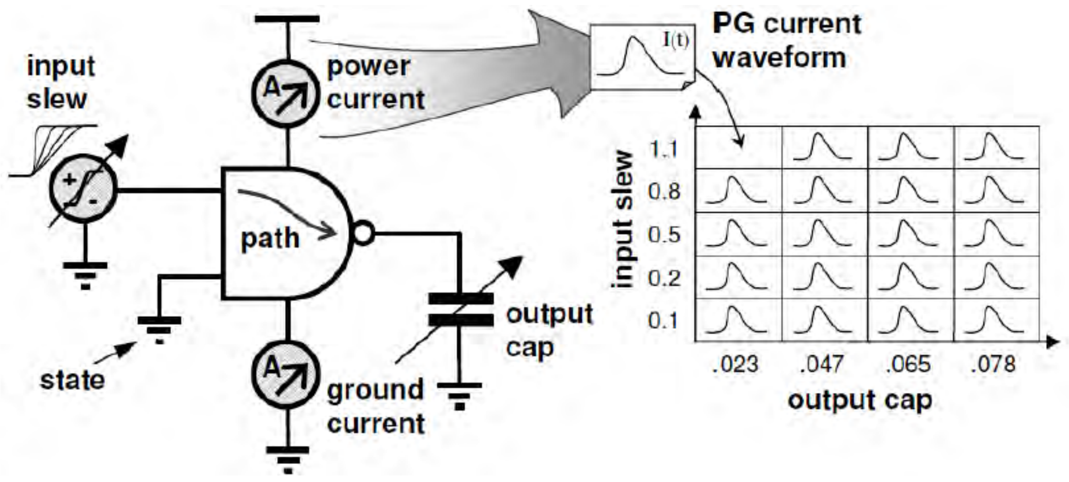
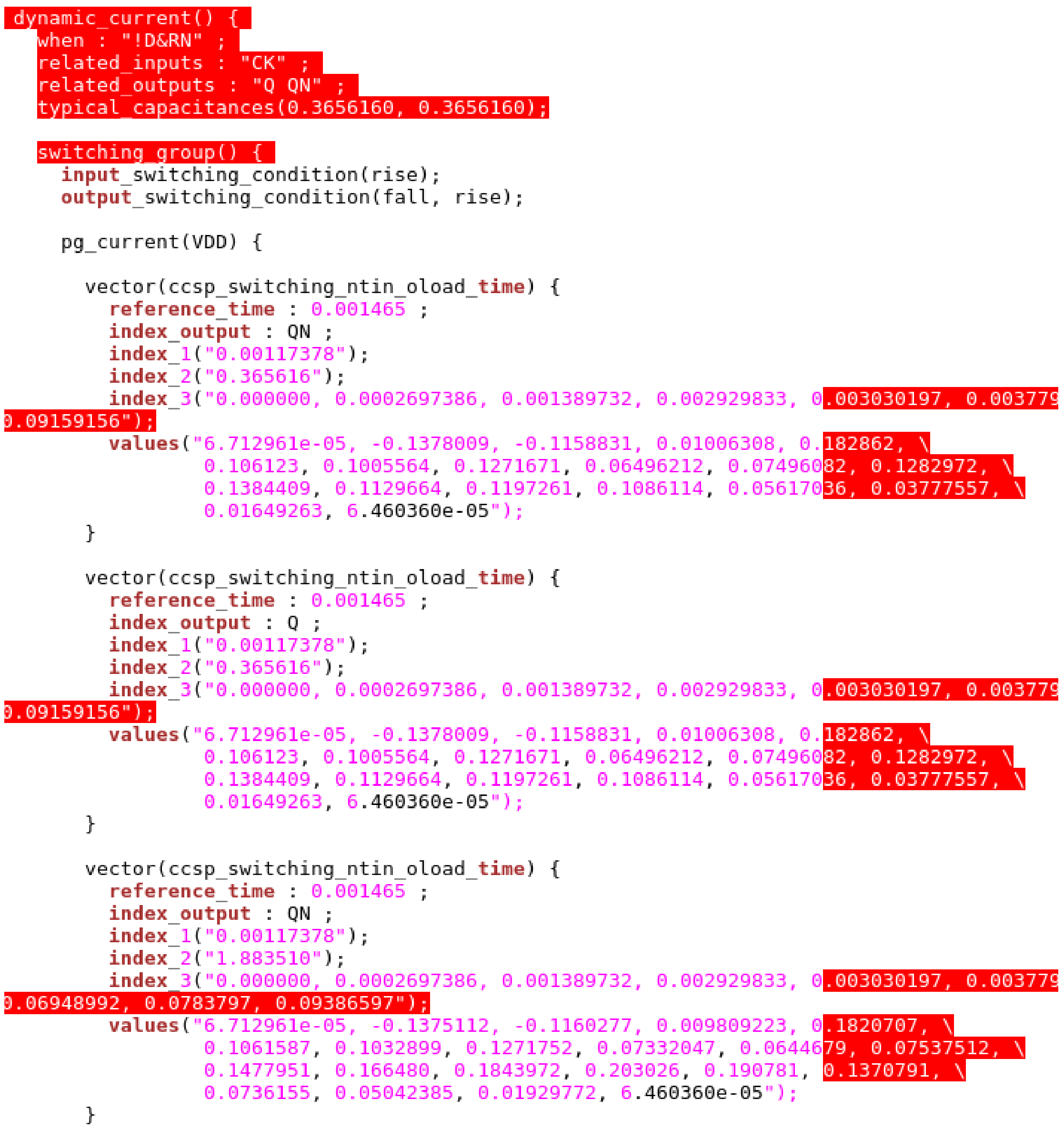


Figure 4.11 Dynamic Current Waveform Measurement

In figure 4.12 we can see an example of dynamic current waveform measurement for a Dflipflop cell.



```
dynamic_current() {
   when : "!D&RN" ;
   related_inputs : "CK" ;
   related_outputs : "Q QN" ;
   typical_capacitances(0.3656160, 0.3656160);

   switching_group() {
     input_switching_condition(rise);
     output_switching_condition(fall, rise);

     pg_current(VDD) {

       vector(ccsp_switching_ntin_oload_time) {
         reference_time : 0.001465 ;
         index_output : QN ;
         index_1("0.00117378");
         index_2("0.365616");
         index_3("0.000000, 0.0002697386, 0.001389732, 0.002929833, 0.003030197, 0.003779
0.09159156");
         values("6.712961e-05, -0.1378009, -0.1158831, 0.01006308, 0.182862, \
                 0.106123, 0.1005564, 0.1271671, 0.06496212, 0.0749608 2, 0.1282972, \
                 0.1384409, 0.1129664, 0.1197261, 0.1086114, 0.05617036, 0.03777557, \
                 0.01649263, 6.460360e-05");
       }

       vector(ccsp_switching_ntin_oload_time) {
         reference_time : 0.001465 ;
         index_output : Q ;
         index_1("0.00117378");
         index_2("0.365616");
         index_3("0.000000, 0.0002697386, 0.001389732, 0.002929833, 0.003030197, 0.003779
0.09159156");
         values("6.712961e-05, -0.1378009, -0.1158831, 0.01006308, 0.182862, \
                 0.106123, 0.1005564, 0.1271671, 0.06496212, 0.0749608 2, 0.1282972, \
                 0.1384409, 0.1129664, 0.1197261, 0.1086114, 0.05617036, 0.03777557, \
                 0.01649263, 6.460360e-05");
       }

       vector(ccsp_switching_ntin_oload_time) {
         reference_time : 0.001465 ;
         index_output : QN ;
         index_1("0.00117378");
         index_2("1.883510");
         index_3("0.000000, 0.0002697386, 0.001389732, 0.002929833, 0.003030197, 0.003779
0.06948992, 0.0783797, 0.09386597");
         values("6.712961e-05, -0.1375112, -0.1160277, 0.009809223, 0.1820707, \
                 0.1061587, 0.1032899, 0.1271752, 0.07332047, 0.06444679, 0.07537512, \
                 0.1477951, 0.166480, 0.1843972, 0.203026, 0.190781, 0.1370791, \
                 0.0736155, 0.05042385, 0.01929772, 6.460360e-05");
       }
```

Figure 4.12 Example of dynamic_current Section

Chapter 5

Characterization Process

The steps that someone has to follow in order to make a complete characterization are described and analyzed in the subchapters below [5].

5.1. Tool Installation

At first, a license must be obtained from Synopsys (or the existing be upgraded) and the tool must be installed following the instructions in the installation file.

5.2. Necessary Files

We must have a number of necessary files in order to be able the tool to run:

- A netlist for each cell to be characterized.
- A reference Liberty model which is not necessary in some flows but we will need it in our (skeleton liberty-based flow).
- A process model library which contains information about the technology of transistors.
- A run.tcl script which calls and provides information to Siliconsmart.
- A configure.tcl script which sets some global parameters to all cells for a given characterization directory. There is a default file in the Siliconsmart installation directory.

All these files must be in our working directory where Siliconsmart will run and create a characterization directory.

5.3. Editing the configure.tcl File

The default file has already the most necessary parameters that we are going to need, so we only have to modify some of them and probably add some more.

The configure.tcl file has several sections in it and includes the following major parameter blocks:

- OPERATING CONDITIONS DEFINITION - This section includes the global parameters that control high-level characterization settings and integration with third-party tools (SPICE, load sharing, and so on).

- GLOBAL CONFIGURATION PARAMETERS - This section includes the global parameters that control high-level characterization settings and integration with third-party tools (SPICE, load sharing, and so on).
- DEFAULT PIN CONFIGURATION PARAMETERS — The parameters in this section control the characterization settings for a class of pins—for example, setting the output load range for a set of pins.
- LIBERTY MODEL GENERATION PARAMETERS — The parameter in this section add Liberty header attributes for use with "*model –create_new_model*". We will not use this command in this thesis so it will not be further analyzed.

The changes that we have to make are:

1. Set the suitable values for vdd, vss and temperature and the location of the process model library in our design with the command (in OPERATING CONDITIONS DEFINITION block):
   set_opc_process op_cond {
   {.lib "/home/location/example.lib" TT}
   }
2. Include the name of operation conditions, that were made by *create_operation_condition ,* in *set_active_pvts {}* command.
3. Select a simulator and a job scheduler as described in the next two subchapters
4. Include the right supply and ground names in the *set power_meas_supplies/grounds {}* commands.
5. Add the command *set model_rise_fall_capacitance 1* through which the SiliconSmart tool will use rise/fall_capacitance Liberty attributes.
6. Add the command *set Model_significant_digits 7* in order to increase the the demical accuracy of the results.
7. Add the command *set gate_leakage_time_scaling_factor 50.0* to scale the total simulation time by 50 times. This is necessary because the simulation time may be not sufficiently long to capture the slowly changing gate leakage current. If a similar warning message is appeared in sis.log file after using this command we have to increase the value to 100.0 etc.
8. Add the command *set auto_fix 0* in order not to let Siliconsmart retry to execute failed tasks again. By this way the characterization is not slowed down.
9. Add the command *set enable_cell_leackage_power_modeling 1.* When enabled, the cell_leakage_power attribute will be modeled for the cell even if the value of the liberty_multi_rail_format is set to v1 or v2. The last eight changes refer to GLOBAL CONFIGURATION PARAMETERS block. Generally we can modify parameters in this section with the *set* command: *set parameter value.*

5.4. Selecting Simulator

The next task is to select the simulator. The available choices are:

- FineSim
- FineSim Multi-CPU Simulation
- FineSim-Embedded which upgrades each characterization engine to include a copy of the embedded version of FineSim SPICE.
- HSPICE
- HSPICE in Client Server Mode
- HSPICE-Embedded

After testing all the simulators we found out that the quickest one is FineSim-Embedded. We must include the followed command in the configure.tcl file:

```
set simulator finesim_embedded
```

Simulator options do not need to be separately defined because Siliconsmart will use the default finesim options.

5.5. Setting a Job Scheduler

The available job schedulers are:

- Stand-Alone
- Load Sharing Facility (LSF) which must be separately installed and set up.
- Sun Grid Engine which must be separately installed and set up.
- RTDA NC

As we do not want to use exterior programs, we use stand-alone scheduler. We must also include two commands in configure.tcl file in order to use this scheduler:

```
set job_scheduler standalone
set run_list_maxsize 8
```

The last command is used in order to run eight jobs simultaneously and fully take advantage of the capabilities of our machine (eight cores).

5.6. Characterization Directory Structure

It is very important to know what there is in the characterization directory that Siliconsmart creates and how are files parceled out. In figure 5.1 we can see the directory structure:

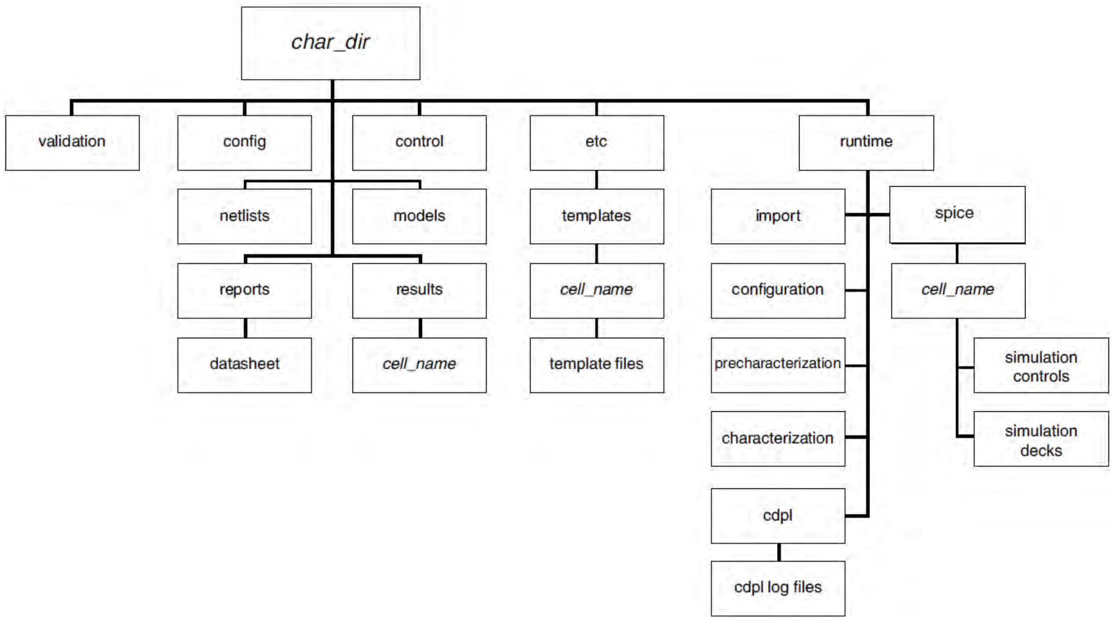


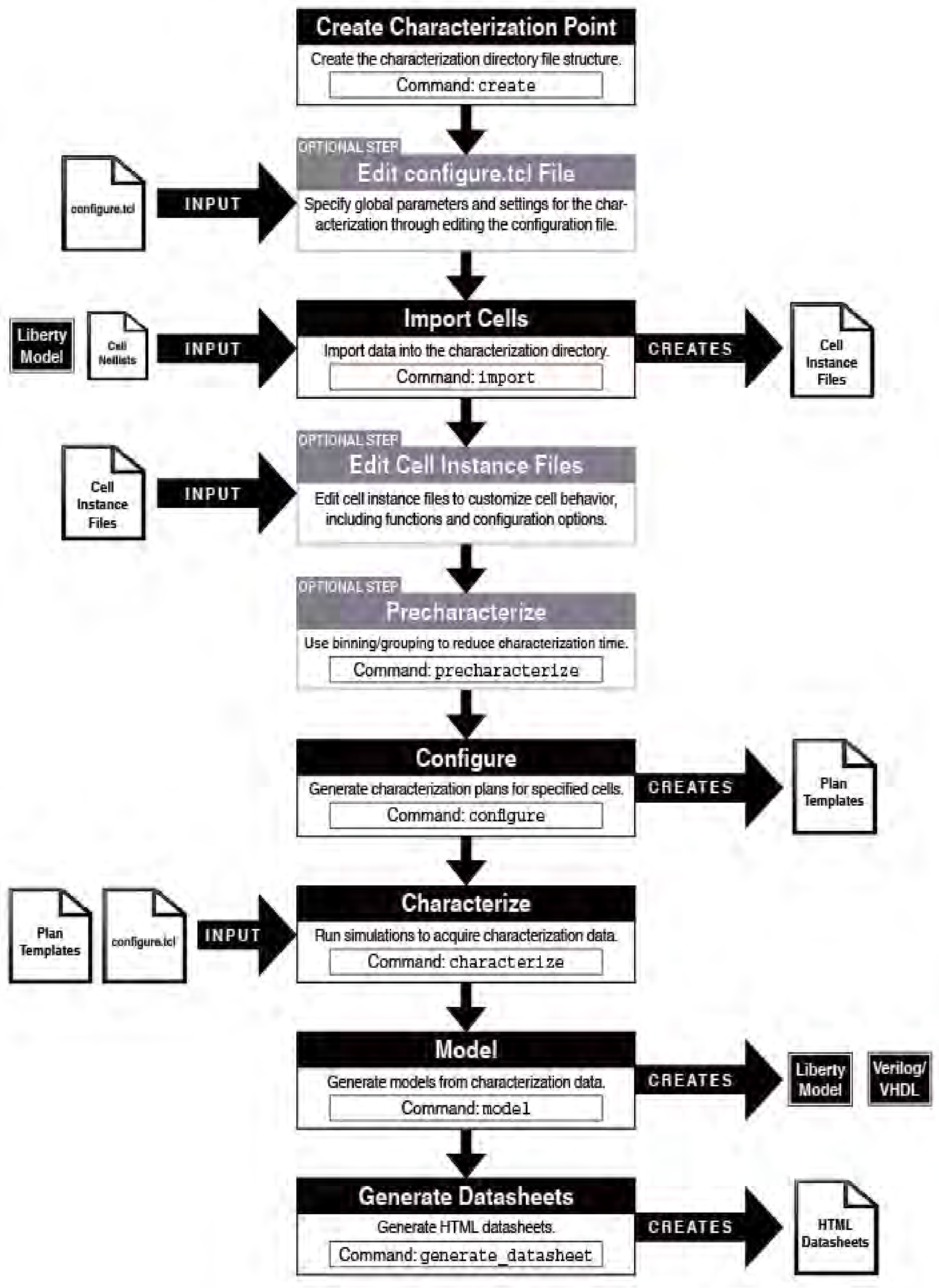
Figure 5.1 Characterization Directory Structure

The most important directories are char_dir/:

- config – It contains the copied configure.tcl script which we use for characterization.
- model – It contains the models that we made after running Siliconsmart.
- results – It is considered the cache storage of current characterization run. Siliconsmart takes characterization plans that has made for each cell and generates the file necessary to describe each measurement to be configured, so each cell has a file.
- validation – It contains the results of validation process which we will not use in this thesis.
- netlists – It contains copies of all the netlists we used.
- report – It contains datasheets for each cell for operating point. The command *model generated_datasheet* is required.
- control – It contains the instance files of the cells. They define the behavior of the cell, including information, function, characterization and modeling configuration options.
- etc/templates – It contains characterization plans for each cell created after configure command. The characterization plan describes each of the measurements to be performed, the stimulus to be applied, and other pertinent information.
- etc/stats – It contains a report for specified cells and precharacterization simulation and it is activated with the command *report_sim_stats*. If nothing is specified, it runs the report for all cells.
- etc/database – It contains the status (done/failed) of each cell and need the *status* command.
- runtime – It contains log files from each cell which help for debugging.

5.7. Siliconsmart Run Process

The final part is to create and run a tcl script which executes the entire characterization flow. This is an automated way to run and re-run the characterization flow which we can see below.



Create Characterization Point
Create the characterization directory file structure.
Command: create
OPTIONAL STEP
Edit configure.tcl File
Specify global parameters and settings for the characterization through editing the configuration file.
configure.tcl
INPUT
Import Cells
Import data into the characterization directory.
Command: import
Liberty Model
Cell Netlists
CREATES
Cell Instance Files
OPTIONAL STEP
Edit Cell Instance Files
Edit cell instance files to customize cell behavior, including functions and configuration options.
OPTIONAL STEP
Precharacterize
Use binning/grouping to reduce characterization time.
Command: precharacterize
Configure
Generate characterization plans for specified cells.
Command: configure
Plan Templates
Characterize
Run simulations to acquire characterization data.
Command: characterize
Model
Generate models from characterization data.
Command: model
Verilog/ VHDL
Generate Datasheets
Generate HTML datasheets.
Command: generate_datasheet
HTML Datasheets

Figure 5.2 Basic Characterization Flow

The structure of the script for the **skeleton liberty-based flow** that we use in this thesis is:

1. Prepare for characterization:
   - Set/create charpoint – The SiliconSmart tool works within a predefined characterization directory structure. Relevant files are expected to reside within this structure and are generated in specific subdirectories.
   - Set log file – This file contains all information printed out from Siliconsmart during characterization. It includes also warnings/errors encountered by the tool, which allows for easy parsing and debugging
   - Copy configure.tcl file in char_dir.
   - Set location – It is used to specify the target characterization directory for all characterization-related data files. The path specified can be absolute or relative and must refer to a valid SiliconSmart directory structure generated by the create command. Furthermore if we modify the configure.tcl file after we copy it in char_dir, we must use this command to re-read it. Also the command *get_location* returns the current char_dir location.

   Commands:

   ```
   set charpoint char_dir
   create $charpoint
   set_log_file $charpoint/sis.log
   exec cp configure.tcl ${charpoint}/config/configure.tcl
   set location $charpoint
   ```

2. Import the following:
   - An existing Liberty model that contains the information you wish to preserve.
   - The netlist.
   - The cells to be recharacterized.

   Command: *import – liberty our_lib.lib –skeleton –extension .spi –netlists/ all*

   -extension switch is used to specify the extension of the netlists.

3. Configure options - The set_config_opt command provides a mechanism for setting global parameters and pin type parameters on a per-cell, per-measurement, or permeasurement-type basis. If we want also to apply some options to a set of cells we must group them with the *set* command and use a name through which we will be able to refer to them (cells).

   Commands:

   ```
   set cells {cell_list}
   set cells2 {cell_list}
   ```

set_config_opt -cells $cells -type type_of_measurement state partitions all
set_config_opt ccs_power_optimize_waveform 0

where:
state_partitions all means that the tool will make measurements for all possible states and
ccs_power_optimize_waveform 0 is used in order to be many time points included to the final waveforms.

4. Configure – This command generates a characterization plan for each cell and prepares it for characterization.

   Command: *configure -timing -ccs -power –ccs_power all*

   We use these swithes to make NLDM, NLPM, CCS-timing and CCS-power characterization.

5. Characterize – It performs the simulations needed to characterize timing, and power for all cells.

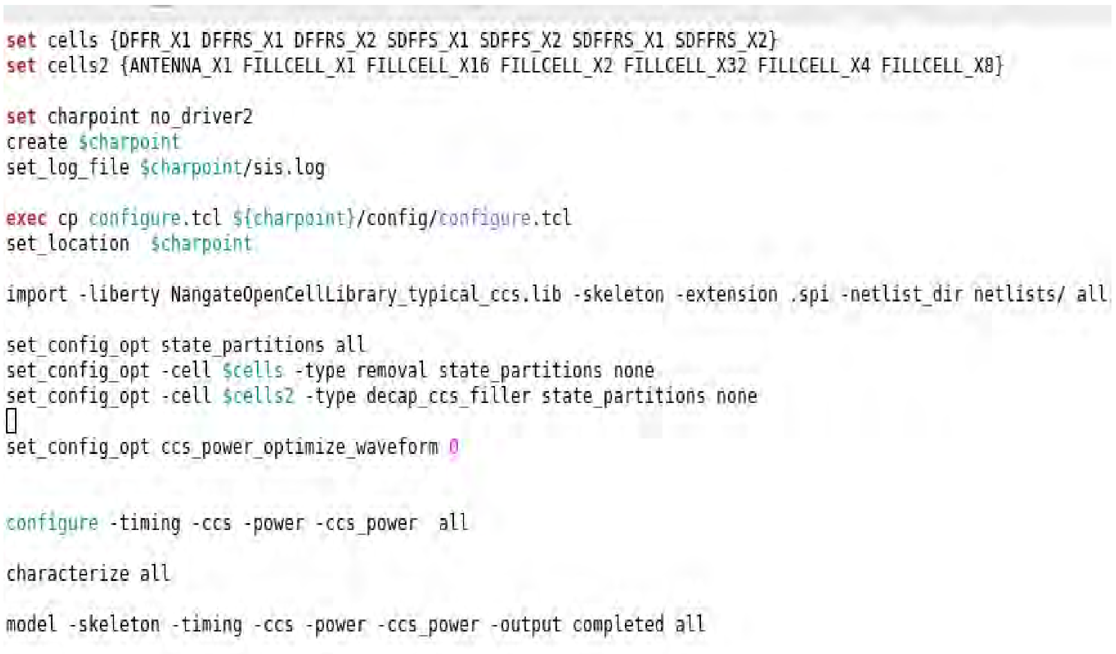
   Command: *characterize all*

6. Model the specified data. When invoked with the -skeleton switch, the Liberty produced will be a as if a brand new Liberty but with all the attributes, custom groups, test_cell, statetable, header, etc., information from the imported Liberty preserved at library/cell/pin-level.

   Command: model -timing -ccs -power -ccs_power -output completed all

   By this way the generated Library will contain all the characterizations that we need and it will be named completed_nameofopcondition.

In figure 5.3 we can see an example flow as it has already described:



```
set cells {DFFR_X1 DFFRS_X1 DFFRS_X2 SDFFS_X1 SDFFS_X2 SDFFRS_X1 SDFFRS_X2}
set cells2 {ANTENNA_X1 FILLCELL_X1 FILLCELL_X16 FILLCELL_X2 FILLCELL_X32 FILLCELL_X4 FILLCELL_X8}

set charpoint no_driver2
create $charpoint
set_log_file $charpoint/sis.log

exec cp configure.tcl ${charpoint}/config/configure.tcl
set_location  $charpoint

import -liberty NangateOpenCellLibrary_typical_ccs.lib -skeleton -extension .spi -netlist_dir netlists/ all

set_config_opt state_partitions all
set_config_opt -cell $cells -type removal state_partitions none
set_config_opt -cell $cells2 -type decap_ccs_filler state_partitions none

set_config_opt ccs_power_optimize_waveform 0


configure -timing -ccs -power -ccs_power  all

characterize all

model -skeleton -timing -ccs -power -ccs_power -output completed all
```

Figure 5.3 Complete Characterization Process

In order to run the script we must, first, change our working directory to the one where all necessary files are (5.2 subchapter) and then execute from the terminal the command *siliconsmart run_script.tcl* to call Siliconsmart.

5.8. Driver Mode

So far we used a simple linear piecewise-linear (PWL) source for driving input transitions which leads to inaccurate models. That is because the shape and slew of the driving waveform significantly affects intrinsic delay and effective input pin capacitance.

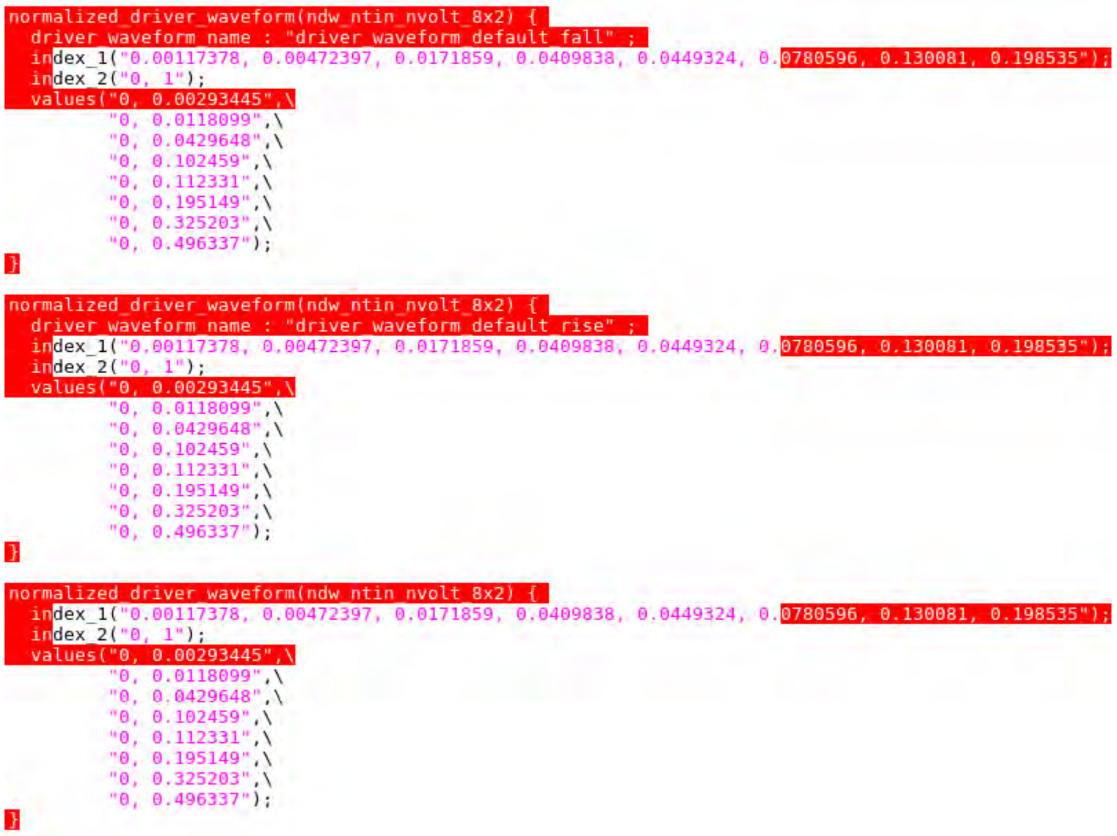
We have to use true active drivers using an actual cell to drive the transition. The SiliconSmart tool requires either a non-inverting buffer cell (or combinational cell configured as a buffer) or an inverter to be used as the active driver. The disadvantage of using an active driver is that it slows down characterization significantly due to the additional driver cell transistors being simulated for every measurement.

In order to solve this problem we use active waveform drivers. This method eliminates the performance problem while still maintaining accuracy. Particularly it recreates the driver waveform and applies it as a multi-point piecewise linear waveform (PWL) which is derived by curve fitting the actual driver cell waveform, and thus achieving significant performance improvements will little or no accuracy penalty.

The implementation of this method is quite simple:

- In DEFAULT PINTYPE PARAMETERS section of configure.tcl file we change the *driver_mode* parameter to *active-waveform* and the *driver* to the cell that we want to be the driver, for example the name of an inverter.
- We can, optionally, use the *driver_waveform_points* parameter which contains a list of floating points number between 0 and 1.0 in ascending order. The default value is *{0.02 0.05 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 0.95 0.98*}. Specifying more points will cause the waveform to be more similar to the actual driver waveform. In general, the defaults for this parameter shows good correlation when compared to active driver mode, so we use this value.
- When we provide your *driver_waveform_points* to be too close,for example *{ 0.05 0.10 0.20 0.30 0.40 0.45 0.5 0.55 0.6 0.7 0.8 0.9 0.95}*, then the *driver_waveform_min_dt* will be activated. The SiliconSmart tool will determine that two voltage points should be at least *driver_waveform_min_dt* interval apart to satisfy the spice engine. If voltage point separation is too close, the second point will be moved so that the separation is at least *driver_waveform_min_dt*, for example *0.5e-13.*
- In our run process tcl script, we have to import the driver through *import_driver cell_name -netlist netlist_file -input_pin pin -output_pin pin.* If we use an inverter we must include the *-inverting* switch to *import_driver* parameter.

In figure 5.4 we can see how is driver waveform modeled.


```
normalized_driver_waveform(ndw_ntin_nvolt_8x2) {
  driver_waveform_name : "driver_waveform_default_fall" ;
  index_1("0.00117378, 0.00472397, 0.0171859, 0.0409838, 0.0449324, 0.0780596, 0.130081, 0.198535");
  index_2("0, 1");
  values("0, 0.00293445",\
         "0, 0.0118099",\
         "0, 0.0429648",\
         "0, 0.102459",\
         "0, 0.112331",\
         "0, 0.195149",\
         "0, 0.325203",\
         "0, 0.496337");
}

normalized_driver_waveform(ndw_ntin_nvolt_8x2) {
  driver_waveform_name : "driver_waveform_default_rise" ;
  index_1("0.00117378, 0.00472397, 0.0171859, 0.0409838, 0.0449324, 0.0780596, 0.130081, 0.198535");
  index_2("0, 1");
  values("0, 0.00293445",\
         "0, 0.0118099",\
         "0, 0.0429648",\
         "0, 0.102459",\
         "0, 0.112331",\
         "0, 0.195149",\
         "0, 0.325203",\
         "0, 0.496337");
}

normalized_driver_waveform(ndw_ntin_nvolt_8x2) {
  index_1("0.00117378, 0.00472397, 0.0171859, 0.0409838, 0.0449324, 0.0780596, 0.130081, 0.198535");
  index_2("0, 1");
  values("0, 0.00293445",\
         "0, 0.0118099",\
         "0, 0.0429648",\
         "0, 0.102459",\
         "0, 0.112331",\
         "0, 0.195149",\
         "0, 0.325203",\
         "0, 0.496337");
}
```

Figure 5.4 Driver Waveform Template

5.9. After Characterization

When Siliconsmart is done with characterization, we should open the sis.log file to check if the flow is correctly executed and then search for warnings and errors that may have occurred. Then, we can open the generated files, for example the Liberty, to check if there is all the expected data in them.

If there is still something missing we can recharacterize the generated library with different configure options.

Chapter 6

Conclusion

In conclusion, given the key role of accurate library models in digital implementation, there has been a significant interest in standard cell library characterization. This thesis reports our effort to conduct a completed characterization of any library used in industry.

Specifically, we focused on the power characterization and especially on CCS power which is incomplete in most open standard cell libraries. For our experiments we used the Siliconsmart tool which is provided by Synopsys and generates accurate model libraries, using Liberty format, for all types of characterization.

In order to run all the characterization flow and give the suitable configuration parameters to the tool, we used tcl scripts. As soon as there are no errors (lack of warnings is also desirable) after characterization, the new extracted libraries can be easily reconfigured and recharacterized.

6.1. Future work

Possible extensions in this project may be the following:

- **CCS noise characterization**

  CCS Noise is an advanced current-based driver model that enables accurate noise analysis with results very close to Spice simulation. It not only precisely models injected crosstalk noise bumps, but also allows more advanced analysis, such as propagated noise bumps and the driver weakening.

- **Process Variation (AOCV)**

  Advanced on-chip variation (AOCV) [10] uses techniques for context specific derating instead of a single global derate value, thus reducing the excessive design margins and leading to fewer timing violations. This represents a more realistic and practical method of margining, alleviating the concerns of overdesign, reduced design performance, and longer timing closure cycles.

- **Integration into an in-house VLSI optimization flow**

  The goal is to be integrated to an in-house VLSI continuous resizing algorithm. Specifically, the algorithm finds the best sizes for each cell, thus an online cell characterization is required for each new cell size which is not part of current library.

Bibliography

[1] NanGate FreePDK45 Open Cell Library, Nangate Inc., 2011
http://www.nangate.com/?page_id=2325

[2] Globalfoundries 40LP, Globalfoundries Inc, version 1.1, 2015
https://www.globalfoundries.com/sites/default/files/product-briefs/pb-40lp.pdf

[3] Siliconsmart Datasheet, Synopsys Inc., 2017
https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/siliconsmart-ds.pdf

[4] https://en.wikipedia.org/wiki/Standard_cell

[5] Siliconsmart User Guide, Synopsys Inc., version M-2017.03, 2017
https://solvnet.synopsys.com/dow_retrieve/M-2017.03/siliconsmart/siliconsmart_olh/index.htm#page/sis_help%2Fcover.htm%23wwconnect_header

[6] Liberty User Guides and Reference Manual Suite, Synopsys Inc., version 2017.06, 2017
https://media.c3d2.de/mgoblin_media/media_entries/659/Liberty_User_Guides_and_Reference_Manual_Suite_Version_2017.06.pdf

[7] CCS Timing Technical White Paper, Synopsys Inc., version 2.0, 2006
https://studylib.net/doc/8869013/ccs-timing-whitepaper

[8] Power Characterization Recommendation for Power Analysis in PrimeTime PX, Synopsys Inc., version 000-1, 2012
https://solvnet.synopsys.com/retrieve/customer/application_notes/attached_files/037594/power_characterization_recommendation_for_PX_v1.pdf?1351837988022

[9] CCS Power Technical White Paper, Synopsys Inc., version 3.0, 2006

[10] Primetime Advanced OCV Technology White Paper, Synopsys Inc., 2009
https://www.synopsys.com/content/dam/synopsys/implementation&signoff/white-papers/PrimeTime_AdvancedOCV_WP.pdf