Library Characterization for Cell-Aware Test

Santosh S. Malagi

CE-MS-2018-27

Abstract

Due to their large number of high-precision, defect-prone manufacturing steps, integrated circuits (ICs) are susceptible to manufacturing defects and hence need to undergo electrical tests to weed out the defective parts and guarantee sufficient outgoing product quality to the customer. A key step in test development for digital logic ICs is automatic test pattern generation (ATPG). Cell-aware test (CAT) is a next-generation test pattern generation approach; its novel feature is that it explicitly addresses cell-internal defects (as opposed to relying on serendipitous coverage by traditional ATPG). CAT consists of two stages - cell-aware library characterization (CA-LC) and cell-aware ATPG. Library characterization uses parasitics-extracted transistor-level netlists to model open and short defects candidates, which are then simulated with an exhaustive set of cell-level test patterns. The results are encoded in the form of defect detection matrices (DDMs). Cell-aware ATPG uses this information to determine a set of test patterns such that, as many as possible cell-internal defects in the circuit are covered.

As an industrial standard-cell library contains hundreds of cells, library characterization is a time consuming task. The target defect set must be realistic and complete, but not unnecessarily large. The objective of this thesis is to improve the library characterization stage of the Cadence CAT flow by effectively and efficiently modelling realistic defects, while trying to minimize the time required for characterization. To achieve this, several improvements to the existing flow are proposed. (1) defining a set of customized settings for the parasitics extraction tool for generating transistor-level netlists, which are well-suited for cell-aware defect modelling (2) elimination of potential defects, which were superfluous elements being inserted into the netlist (3) using super-hard defect resistance values for modelling opens and shorts (4) reduction in simulation time by modifying the software flow and, (5) inserting a single short defect between two net pairs to reduce the size of the target defect set. For the 45nm generic library (GPDK045) from Cadence, these modifications resulted in an improvement in test quality by uncovering as many as 1114 false detections and a reduction of 6% in the characterization time. The number of short defects to be simulated reduced by 97.7%. This work was carried out as a part of a joint project on cell-aware test between Cadence (supplier of electronic design automation software), IMEC (research organization), and Eindhoven University of Technology.

Library Characterization for Cell-Aware Test

by

Santosh S Malagi

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Engineering

at the Delft University of Technology,
to be defended publicly on Tuesday September 18, 2018 at 1:00 PM.

Student number:	4602420	
Supervisor:	Prof. dr. ir. Said Hamdioui,	
Thesis committee:	Dr. ir. Rene van Leuken,	TU Delft
	Ir. Erik Jan Marinissen,	IMEC

Delft University of Technology

Dedicated to the worldwide community of DfT and EDA engineers, who strive to make the world a better place, one chip at a time!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ATPG	Automatic Test Pattern Generation
CAT	Cell-Aware Test
CA-ATPG	Cell-Aware Automatic Test Pattern Generation
CA-LC	Cell-Aware Library Characterization
CUT	Circuit Under Test
DfT	Design for Test
DDM	Defect Detection Matrix
EDA	Electronic Design Automation
IC	Integrated Circuits
PEX	Parasitics Extraction
PVS	Physical Verification System
QRC	Quantus RC extraction tool
RAK	Rapid Adoption Kit

ACKNOWLEDGEMENTS

This thesis is a result of the support and the guidance I received from several people, and I am forever indebted to them. I consider myself to be fortunate to be associated with such individuals and organizations. Foremost, I would like to express my sincere gratitude's to my thesis advisor Prof. Said Hamdioui, who initiated me into the world of VLSI Testing and DfT with his inspiring lectures and magnetic personality. Without him, I don't think I would have ventured into this discipline or thought of making a career as a DfT engineer. If Prof. Hamdioui showed me the way my daily supervisor at IMEC, Erik Jan Marinissen taught me how to walk on this path. Erik Jan refined my style of working, corrected me when I was wrong, and showed me the importance of writing correctly and making effective technical presentations. The lessons I learned from him during this short tenure at IMEC will go a long way in shaping my professional career. My sincere thanks to my colleague Zhan Gao, for her untiring support and interesting discussions. I am sure, Zhan will make a good researcher and I wish her the very best in her future endeavors.

Next I would like to extend my acknowledgements to Kristof Croes who not only funded my project but taught me that respect has to be earned, and it takes time and effort to reach great heights. Prof. Ingrid De Wolf from KU Leuven, was an able mentor during this journey. Without the cooperation of IT support staff at IMEC, especially my Linux Guy - Ibrahim Tatar my software would never have worked. Thank you Linda Vanmeerbeek, for providing a nice office working space and the colourful marker pens!

Joe Swenton, Principal Software Architect, and Carl Wisnesky, Product Manager, from the Test Solutions Group at Cadence were my software gurus. This was my first real world software development experience both Joe and Carl took keen interest in me, and mentored me. Anton Klotz, University Program Manager at Cadence EMEA sent me a powerful laptop to work on, took care of all the administrative formalities and was kind enough to extend his professional network for my advantage. A special mention of Michel Montanuy - Staff Application Engineer from Cadence France, who taught me two industry leading software tools - Quantus QRC and PVS. As an organization, Cadence was instrumental in providing me with all the tools necessary to find the answers I needed during the course of this project. Not to forget, Vivek Chickermane and Louis Milano at Cadence, for offering me a suitable position to continue my journey with Cadence after my graduation!

The Delft University of Technology, and the people there especially my master coordinator, Prof. Arjan van Genderen, CE group secretaries, Joyce van Velzen and Lidwina Tromp. Prof. Rene van Leuken for consenting to being a part of my thesis exam committee. My fellow batch-mates from TU Delft Aritra Sarkar, Prashanth G L, Sarthak Sharma, Manoj Payani, Srinidhi Ramdas, and Sunil Suresh, many cheers to our friendship! The international student community at the Wisteria Student Residence in Leuven, for all the fun during the thesis period.

None of this would ever have been possible if not for my family, my father for believing in me and giving away his life's fortune for my education, my mom for letting me go, and my sister for her sacrifices.

Santosh S Malagi
Leuven, September 2018

1 INTRODUCTION

As Integrated Circuits (ICs) become pervasive in all walks of life, it is necessary to ensure they meet high quality standards and are reliable. Section 1.1 begins by discussing the motivation behind IC manufacturing test and its importance in electronic product development life-cycle. This is followed by a discussion about the traditional approaches to manufacturing test pattern generation and their inability to model realistic defects based on actual physical layouts. Cell-Aware Test (CAT) is a next generation test pattern generation technique which can address these shortcomings. Section 1.2 provides an overview of the cell-aware test methodology and it's associated challenges. Section 1.3 highlights the major contributions of this thesis, followed by the organization of the rest of the report in Section 1.4.

1.1. MOTIVATION BEHIND IC MANUFACTURING TEST

ICs are omnipresent today; not only in computers, consumer electronics, and smartphones, but increasingly also in the Internet of Things (IoT), automotive and healthcare. Due to their large number of high-precision, defect-prone manufacturing steps, ICs are susceptible to manufacturing defects and hence need to undergo electrical tests to weed out the defective parts and guarantee sufficient outgoing product quality to the customer. High product quality and reliability become an absolute necessity, especially for applications such as automotive and medical which cannot tolerate any defective chips. However, as every IC is individually tested (typically even twice - once at the wafer level and once more after packaging), the tests also need to be very efficient with respect to the execution time. If the cost of identifying a defective chip is $1, it costs about $10 to detect the same defective chip when mounted on a PCB. The penalty rises to $100 when the chip becomes a part of a bigger electronic system. Therefore, the goal of IC testing is to weed out defective chips in the supply chain as early as possible. No amount of testing can guarantee a 100% defect free product, but what testing does for us is, it increases our confidence in the correct and reliable operation of the Circuit Under Test (CUT). In volume production only those chips which pass the test are shipped to the customer and the rest are discarded.

1.2. CELL-AWARE TEST

A key step in the test development for digital logic ICs is automatic test pattern generation (ATPG). ATPG determines the content of the IC test. The quality of the generated test patterns significantly affects the test effectiveness. For scan-testable digital logic, manufacturing test patterns can be automatically generated by ATPG software tools. Conventional ATPG tools operate under the assumption that defects can only occur at the periphery of standard-cell instances, I/O ports or between interconnect lines. Any defect occurring within a standard-cell is completely disregarded during test pattern generation. As semiconductor technology scales further and newer transistor architectures are introduced, it is expected that the percentage of defects which occur within the standard cells i.e cell-internal defects will increase considerably [1–3]. Therefore, test patterns must be capable of detecting such cell-internal defects. Cell-aware test is a next-generation test pattern generation methodology. Its novel feature is that it explicitly addresses cell-internal defects as opposed to relying on serendipitous defect coverage reported by traditional ATPG tools. Cell-aware test is expected to significantly improve the test quality, and hence reduce the test escape rate for a relatively small increase in test execution cost. Test generation in cell-aware test flow consists of two distinct stages:

1. *Cell-aware library characterization (CA-LC):* Library characterization begins with the extraction of transistor-level netlists along with the parasitic resistors and capacitors from the standard-cell layouts. Parasitic resistors and capacitors form the basis for identifying possible locations for open and short defects. This is followed by detailed simulations to determine the cell-level test patterns which can detect such intra-cell defects. The results are finally encoded in the form of a defect detection matrix (DDM).

2. *Cell-aware automatic test pattern generation (CA-ATPG):* Based on the generated DDM's, cell-aware ATPG generates minimum chip level test pattern set, which can cover as many as possible cell-internal defects.

Challenges in Cell-Aware Test
As an industrial standard-cell technology library easily contains about 1,500 cells, library characterization is a challenging task. For every cell with n-inputs all possible defect locations within the cell must be identified and simulated with 2^n input test patterns, resulting in a nested loop as illustrated in Figure 1.1. The characterization process therefore is both resource and compute time intensive. Fortunately, it has to be done once per every standard-cell library and the results of this step can be re-used multiple times for all designs based on the same technology library.

```
for all cells ε listofCells do {
        RC parasitics extraction;
        defect candidate identification;
        for all defect candidates d do {
            for all input patterns p do {
                if Simulation(c,p) ≠ Simulation (c+d,p):
                    $DDM_c(d,p)$ = "1"  //defect is detected
                else:
                    $DDM_c(d,p)$ = 0  //defect is not detected
            }
        }
}
```

Figure 1.1: Three nested loops in library characterization.

The cell-aware test flow uses parasitic resistors and capacitors to model possible locations for open and short defects. If all extracted resistors and capacitors are modelled as defect candidates, then the target defect set becomes very large. The parasitic resistors and capacitors to be chosen for cell-aware defect modeling must be such that the defect locations are realistic. Too few implies the risk of missing defects, and too many means expensive simulations and execution time. Since cell-aware test is a non-standard application of parasitic extraction (PEX) tools, conventional approaches of using PEX tools may not be well-suited for cell-aware test. Therefore, an investigation is needed for defining the suitable settings to be used by a PEX tool for generating transistor-level netlists appropriate for cell-aware defect modelling.

1.3. CONTRIBUTION OF THE THESIS

The objective of this thesis is to improve the library characterization stage of the CAT flow based on Cadence EDA tools. It proposes several approaches to effectively and efficiently model realistic physical defects based on standard-cell layouts, while trying to minimize the time required for characterizing standard-cell libraries. The main contributions of this thesis are the following:

- Proposing a set of customized settings for the Quantus parasitics extraction tool. These settings are effective in generating transistor-level netlists which are well-suited for cell-aware defect modelling.

- A heuristics based approach for defining the threshold values for parasitic resistors and capacitors to decide whether they are to be considered as possible defect candidates.

- Eliminating potential defects, which were superfluous elements being introduced in the netlist as a part of the original flow. Due to this change, 1114 false detections were uncovered leading to an improvement in the quality of the defect simulation results for the 45nm library (GPDK045) from Cadence.

- Using super-hard defect resistance values for modelling opens and shorts resulted in an additional 99 defect detections. The simulation time required for characterizing the library cells dropped by 6%.
- Concatenating the defect-injected netlists into one large file and avoiding repeated execution of background software tasks results in a speedup of upto 12 times in simulation time.
- Modifying the short-defect insertion strategy to insert a single short between two net pairs. Consequently simulation of equivalent defects exhibiting the same defect behavior is avoided. This reduces the number of short defect candidates for the GPDK045 library by 97.7%. Once incorporated in the software flow, this change is expected to translate to a significant savings in simulation time.

1.4. THESIS ORGANIZATION

The remainder of the thesis is organized as follows. Chapter 2 provides a background on the essentials of IC testing, standard-cell based design flow, operation of parasitic extraction tools and Quantus QRC based parasitics extraction flow. This sets the tone for rest of the thesis. Chapter 3 presents a detailed discussion on cell-aware test and it's two stages. Chapter 4 talks about the shortcomings in the existing Cadence cell-aware flow and how it affects the quality of defect simulation results. Chapter 5 suggests various modifications to address these issues. Chapter 6 talks about the expected improvements and experimental results obtained by applying these changes to the the existing Cadence cell-aware flow. Finally, Chapter 7 concludes the thesis.

2 Background Knowledge

This chapter equips the reader with the necessary background required to understand the rest of the thesis. Section 2.1 introduces the essentials of IC testing. Section 2.2 touches upon the various aspects of the standard-cell based design flow, which has become the de-facto standard for digital ICs. Parasitics extraction generates a transistor-level netlist which is to used for cell-aware defect modelling. An overview of parasitic extraction is covered in Section 2.3, followed by the specific details of the Quantus QRC parasitics-extraction tool in Section 2.4.

2.1. Introduction to IC Testing

The last 40 years have been exciting times for the semiconductor industry, starting from a single transistor we have reached a stage where billions of transistors are packed in a tiny chip. This has been possible due to the steady decrease in the feature size of transistors and interconnect dimensions. Manufacturing chips at such a level of complexity involves hundreds of process steps, this increases the possibilities of a small error to result in a defective chip. The objective of IC testing is to separate the defective chips from good ones and ensure sufficient outgoing product quality to the customer. This basic philosophy behind IC manufacturing test is depicted in Figure 2.1.

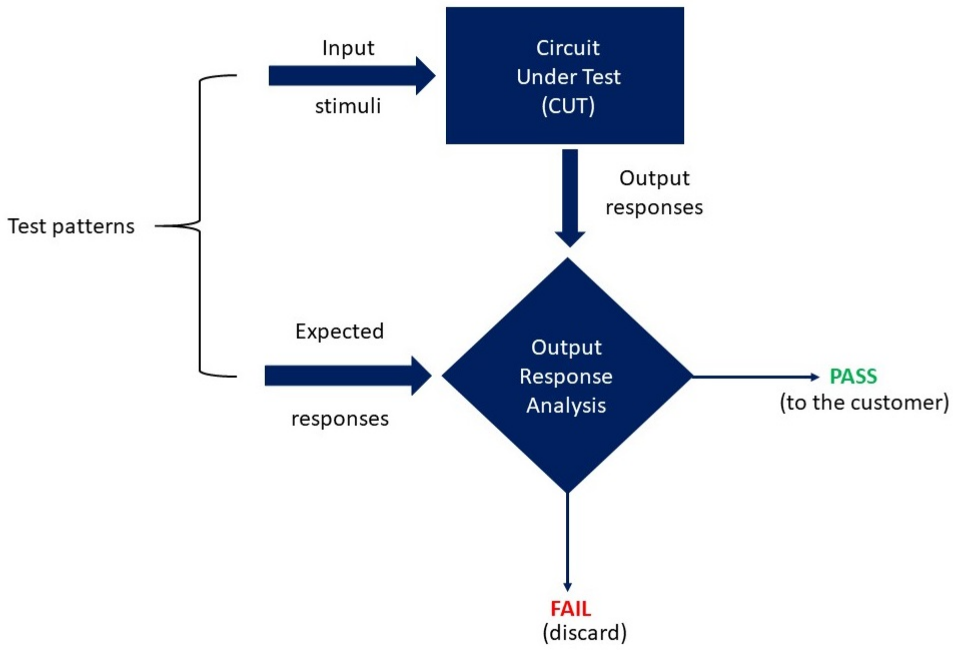


Figure 2.1: Concept of IC manufacturing test.

Input stimuli are applied to the *circuit under test (CUT)* during manufacturing test. If the output response matches the expected response, the circuit (chip) passes the test and is considered to be *fault-free*. Those chips which do not produce the expected response are considered to be *faulty*, and discarded. A *defect* is defined as a physical imperfection or anomaly which results in a faulty behaviour. Some of the common examples of manufacturing defects include: resistive bridges and opens, partially filled vias, missing contacts, process variations, and impurity induced defects. Manufacturing tests can be executed based on two paradigms - *functional testing* and *structural testing*. In *functional testing*, an n-input circuit under test is subjected to all possible (2^n) input combinations to verify it's functionality. Such an approach is not feasible because of the explosion in the number of test patterns required and prohibitive test application times. A more practical approach is *structural testing*. The structural testing approach does not attempt to verify the functionality of the circuit, rather it tries to ascertain whether the circuit has been assembled correctly using low-level building blocks and they have been connected as expected. The quality of the manufactured chips is measured in terms of the Defective parts per million (DPPM) i.e the number of defective chips per million manufactured chips. Certain applications such as automotive target zero DPPM. Manufacturing *yield* is defined as that percentage of chips which pass the manufacturing test and are acceptable, out of the total lot of manufactured chips. *Yield-loss* is that fraction of good chips which are erroneously flagged to be defective. Yield loss occurs because of over testing or badly designed test strategy. The defective chips which are passed to the customer as good ones are known as *test-escapes*.

2.1.1. FAULT MODELLING

Due to the infinite possibilities in which manufacturing defects can manifest themselves, it is highly impossible to test for each one of them individually. Rather we abstract the defects in terms of their faulty behaviour and generate test patterns to detect them. A *fault model* can be defined as a behavioural representation of the actual physical defect. A variety of fault models, each targeting a particular class of defects have been proposed. A good fault model must not only mimic the behaviour of the actual physical defect, but must be computationally efficient and lend itself for test pattern generation. In most cases a single fault model may not be sufficient to meet the targeted test quality. Therefore, a combination of two or more fault models are often used for generating tests. Figure 2.2 depicts some examples of fault models defined at various levels of abstraction.

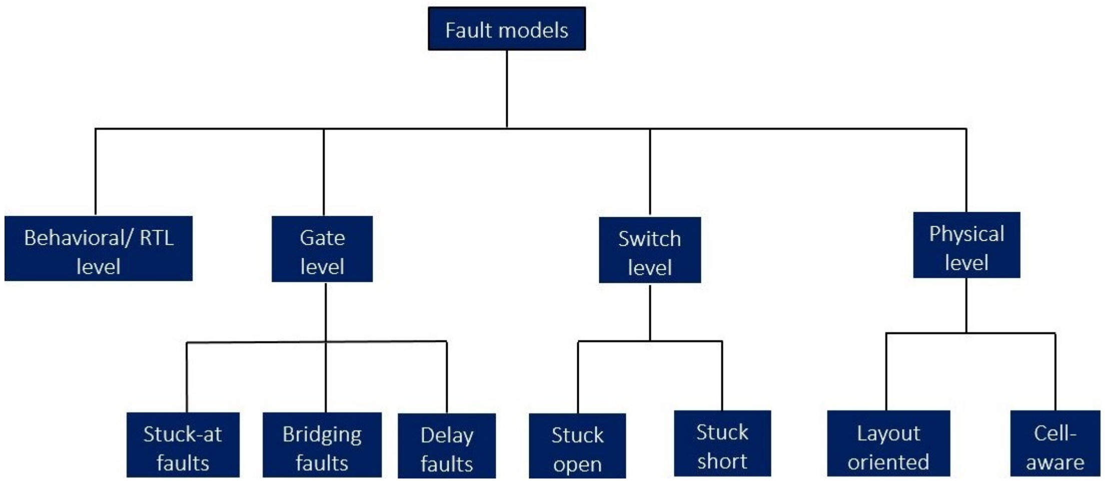


Figure 2.2: Examples of fault models at various levels of abstraction.

Most ATPG tools are based on the use of gate-level fault models such as stuck-at, bridging, and delay faults for generating test patterns. A stuck-at fault model operates on the principle that any interconnect line, *primary input* or *primary output* could be permanently set to a logic-1 value *(stuck-at 1)* or a logic-0 *(stuck-at 0)*. In Figure 2.3 there are nine possible fault sites and each fault site could be stuck-at 0 or stuck-at 1. For single stuck-at fault model based test generation, if two or more faults exhibit the same fault behaviour they are said to be *equivalent*. Rather than testing for all faults in the equivalent set, it suffices to test for only one fault and all other cases are automatically covered. In the above example, for a two-input AND gate a stuck-at 0 fault at the input is same as a stuck-at 0 fault at the output. Similarly, for a two-input OR gate a stuck-at 1 fault at the output is equivalent to stuck-at 1 fault at one of the input lines. The process of substituting several faults with one equivalent fault is known as *fault collapsing*. Fault collapsing helps reduce the length of the test pattern set and time required for fault simulation. *Fault coverage* is a measure of test effectiveness, and is defined as the percentage of faults which are detected by applying test patterns. A bridging fault is used to model the case of two signal nets being shorted with each other. A delay fault models a defect which does not change the value of signal, but causes a circuit to produce the correct output at slower clock speeds. A high-to-low or a low-to-high transition is required to activate the fault effect and hence a pair of test vectors are necessary to detect delay faults.

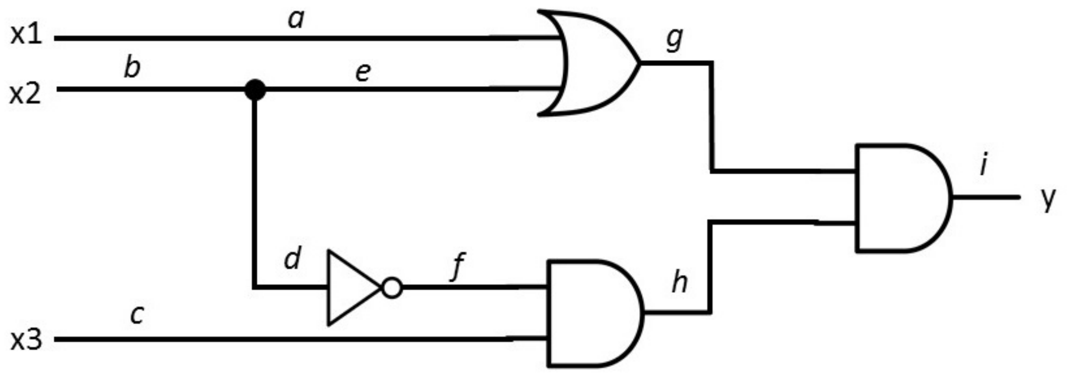


Figure 2.3: Circuit to illustrate concept of fault sites.

A switch level fault model targets transistor faults wherein a transistor could be permanently turned ON - stuck-short (stuck-ON), or turned OFF - stuck-open (stuck-OFF). For detecting stuck-open faults a pair of test vectors are necessary, whereas to detect stuck-short faults the power supply current at steady state is monitored using I_{ddq} testing. Gate-level fault models lie at the intersection of logic and physical implementation and they have been widely used for generating manufacturing test patterns. Unfortunately, such fault models may not be sufficient to address the testing challenges posed by today's state of the art deep sub-micron technologies [1, 4]. Generating test patterns by considering the actual physical layouts to target realistic defects is becoming increasingly important. One such example is the cell-aware model which forms the basis of this thesis work.

2.1.2. FAULT SIMULATION

A circuit simulation is used to predict the behaviour of a circuit. A distinction between functional simulation and fault simulation is necessary at this point. In functional simulation the main intent is design verification. Functional simulation identifies design errors by comparing the circuit responses with specifications or requirements of the design. Functional simulation can take place at various levels of abstraction. For example, a simulation to verify behaviour of the gate-level netlists is often known as logic simulation. At the switch level, analog simulations on SPICE netlists can be carried out to verify the values of functional parameters of the circuit such as output voltage and currents. Carrying out detailed analog simulations of the circuit is usually time consuming, gate-level or logic simulations are much faster. Fault simulation on the other hand characterizes the circuit behaviour in the presence of faulty circuit elements. The goal is to identify a set of test vectors which can detect these faults. It helps estimate the quality of the test pattern set in terms of the targeted fault coverage. Fault simulation results also play a major role in fault diagnosis. Before discussing fault simulation techniques, it is necessary to develop an appreciation of the various logic symbols used in fault simulation. Most fault simulation methods are derivatives of logic simulation techniques and hence use a similar etymology.

Logic Symbols - 0, 1, X, and Z

Boolean algebra uses two logic symbols - 0 and 1 to represent false and true values respectively. In CMOS technology a 0 corresponds to voltage level V_{ss} and 1 represents the V_{dd} value. In addition two more logic symbols X (unknown state) and Z (high-impedance) are used. The symbol X is used to represent unknown logic values on a signal net, when it is not clear whether the value is a logic 0 or logic 1. The outcomes of performing fundamental boolean operations on $0, 1$ and X are depicted in Table 2.1. It can be seen that during a logical AND operation, 0 is the dominant value, whereas 1 is dominant during a logical OR.

Table 2.1: Logic operations on 0,1, and X.

NOT	**0**	1	x		**OR**	**0**	**1**	**x**		**AND**	**0**	**1**	**x**
	1	0	x		**0**	0	1	x		**0**	0	0	0
					1	1	1	1		**1**	0	1	x
					x	x	1	x		**x**	0	x	x

The logic symbol Z means a high-impedance state, it represents a node which is floating i.e. it is neither being driven by a logic 1 nor logic 0 value. Consider the example circuit shown in Figure 2.4. If because of a certain fault, all the drivers are disconnected from the node y, it is said to be in the high-impedance or Z state. Since the internal node y is in high impedance state, the final output f is unknown, it can either take a logic-1 or logic-0 value and is represented using an X symbol.

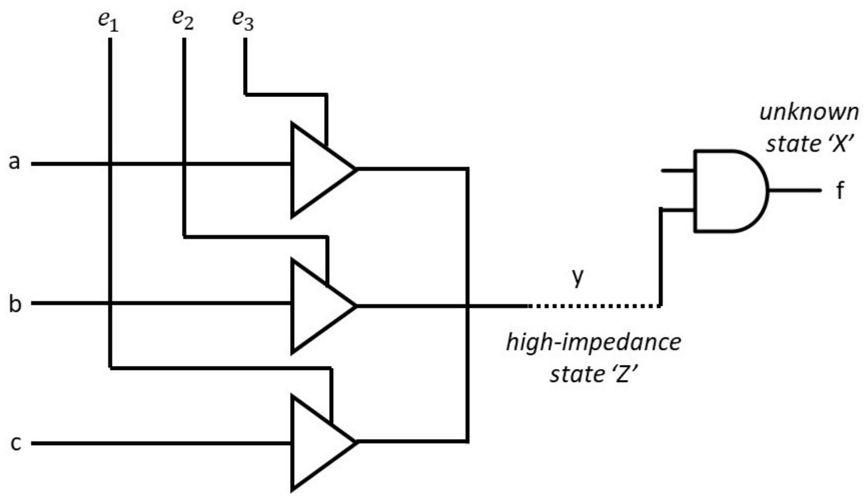


Figure 2.4: Illustration of unknown and high-impedance states.

Various techniques to improve the performance of fault simulation have been proposed in literature [1, 5], and are widely used in practice. The simplest of all the fault simulation techniques is serial fault simulation. A target set of faults is identified and faulty circuit elements are injected in the fault-free circuit netlist. Each of these fault-injected netlists are subject to fault simulation by applying input test vectors. The responses of the fault injected netlists are compared with the expected responses, in case of a mismatch the fault is considered detected. Serial fault simulation is very intuitive and simple to implement. A major drawback however is its slow speed. Parallel fault simulation exploits the bit wise parallelism in boolean operations to reduce the simulation time. Faulty circuits can be simulated in parallel and their responses are compared with that of the fault free circuit. Concurrent fault simulation or event driven simulation only simulates that region of the circuit which is in the immediate vicinity of the fault, thereby reducing the fault simulation time [5]. The choice of a particular simulation technique depends on various factors such as - simulation time, available compute resources, ability to handle unknown or high-impedance states and the structure of the circuit netlist itself [1].

2.1.3. AUTOMATIC TEST PATTERN GENERATION

Given a set of faults F and a set of test patterns T, the objective of ATPG is to determine an optimum subset of test patterns V which can either detect all faults in F, or pre-determined fraction of the faults (i.e. targeted fault coverage). ATPG is a well-known NP-complete problem [6]. ATPG is regarded as a huge binary decision tree, which could have an exponential search space with single, many or no solutions at all [1, 6]. The usual approach to test pattern generation is to first start by generating random test patterns and then apply deterministic test patterns to improve the fault coverage. Many easy to detect faults can be detected using random test pattern generation. After subsequent iterations when fault coverage using random test pattern generation finally saturates, test patterns are generated using deterministic algorithms such as D [7, 8], PODEM [9] and FAN [10]. ATPG algorithms operate on the principles of *sensitizing (activating)* the fault effect, *propagating* the fault effect to the primary outputs, and finally *justifying* all input signals by setting them to *non-controlling* states so that they do not change the faulty logic values. Fault activation sets the signal driving the fault site to a logic value which is opposite to that of the faulty value. For example, to test for a stuck-at 1 fault, the fault site is set to a logic-0 value to activate the fault. Fault propagation selects a path from the fault site to the primary output, where the fault effect can be observed. Fault justification sets the internal nets or primary inputs to non-controlling values so that they do not negate the fault effect. Conflicts are possible between the propagation and justification tasks, in such a scenario a new alternative path must be selected. Consider a simple logic circuit illustrated in Figure 2.5 which has a stuck-at 0 fault at *b*. Let *e*-*g*-*i* be the chosen path for propagating the fault effect to the output *i*. To sensitize the fault effect, *b* is set to logic-1. *a and c* must be justified by setting them to non-controlling values of 0. After completion of the activate, propagate and justify tasks the test pattern becomes *x1 = 0, x2 = 1, and x3 = 0.*

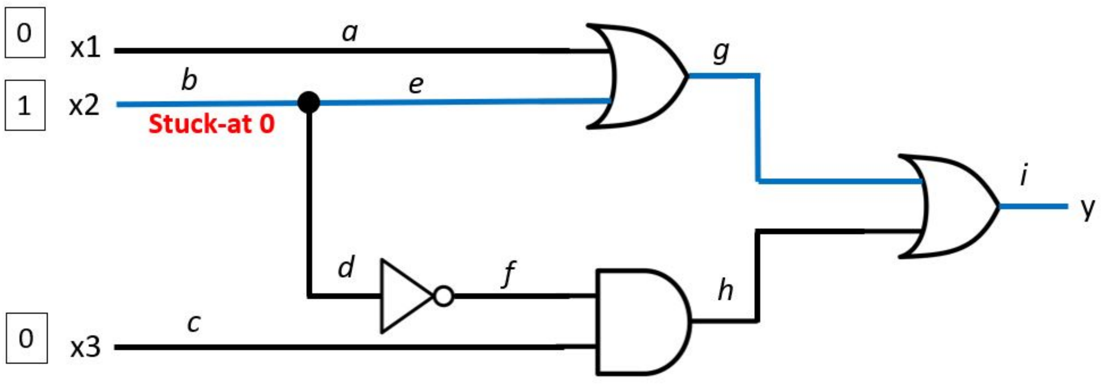


Figure 2.5: Fault sensitization and propagation.

2.2. STANDARD-CELL BASED DESIGN FLOW

A design flow can be defined as set of well-defined steps which allows a chip design team to realize a successful implementation of chip starting from it's specifications. The standard-cell based design flow is widely used for designing Application Specific Integrated Circuits (ASIC's) with substantial digital and memory content. A standard-cell is a complete logical and physical implementation of most commonly used logic or storage function such as AND, OR, NOT gates, multiplexers, adders, flip-flop, register etc. These functions are verified, designed and laid out in a particular technology node and the logical, timing, physical, and electrical models are made available. A collection of such standard-cells in a particular technology node is called as a standard-cell library. A highly simplified version of standard-cell based design flow is illustrated in Figure 2.6. In practice the design flow is not very straight forward and consists of several design iterations and back-tracking steps to resolve design errors. The design flow starts with the system-level design. The detailed behavioural specifications of the chip are captured using a high-level modelling language such as SystemC. RTL designers then use hardware description languages such as VHDL or Verilog to achieve the desired functionality in terms of memory elements (registers) and logical/arithmetic operations performed on them. Functional verification confirms that the developed RTL code matches the design specifications. During the logic synthesis phase, the RTL description is transformed to a technology dependent gate-level netlist. This is done by mapping the logic functions to their respective standard-cell instances in the chosen technology library. This marks the end of front-end design or logic design phase. Additional logic is inserted to improve the testability of design. DFT features make it easy to develop and apply test patterns to detect any defects during manufacturing. Manufacturing test patterns are generated using ATPG tools.

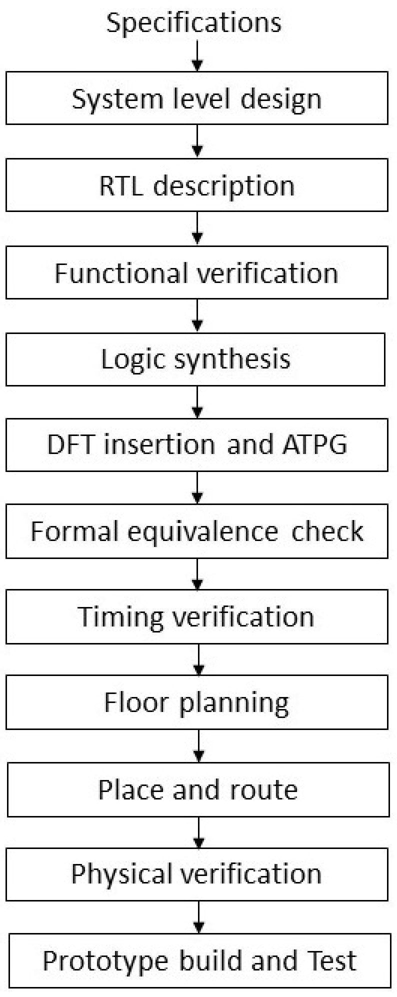


Figure 2.6: Standard-cell based design flow.

The gate-level netlist from the logic synthesis step is modified during DFT insertion. To verify that such a netlist does not violate the initial specifications a formal equivalence check is necessary. The goal of functional verification is to check for functional correctness, where as formal verification checks whether two gate-level implementations of the circuit represent the same boolean equations or not. During pre-layout Static Timing Analysis (STA), the design is checked for setup and hold violations before being handed over to physical design implementation. The idea is to check whether the design can operate at the designated clock frequency. The floor planning stage decides the I/O structure of the chip, placement of standard cell instances and macro blocks, and the design of power-ground networks. The actual placement of standard-cell instances and other blocks happens during placement. For digital designs this step is largely automated and determines the quality of routing. This is a key step in the physical design implementation phase as placement of blocks directly impacts the quality of routing. The clock network is designed during the clock-tree insertion phase. This is followed by routing of signal nets. Global routing determines the 2D map of how standard-cell instances must be interconnected. Detailed routing stage determines the exact interconnection path through different metal layers and vias. The physical verification phase consists of many steps. Design Rule Check (DRC) ensures that the layout does not violate any design rules set by the fabrication house and hence can be manufactured. Layout vs Schematic (LVS) ensures that post layout the structure of the chip has not been modified and matches the circuit schematic. This is followed by circuit extraction step which extracts the designed and parasitic devices (RLC) from the layout. The design is then verified in the presence of actual parasitic elements and wire delays for timing correctness during post-layout STA. Once the back-end flow is complete, the GDSII output is generated and sent for fabrication. An initial batch of chips is subjected to post-silicon validation to ensure all the specifications are satisfied. Any anomaly detected in this stage may lead to back-tracking and corrections in the earlier steps. A chip design project reaches a logical conclusion when the design is cleared for volume production. Every chip is then subjected to manufacturing test by applying test patterns. Only those chips which qualify the tests are shipped to the customer and the rest are discarded.

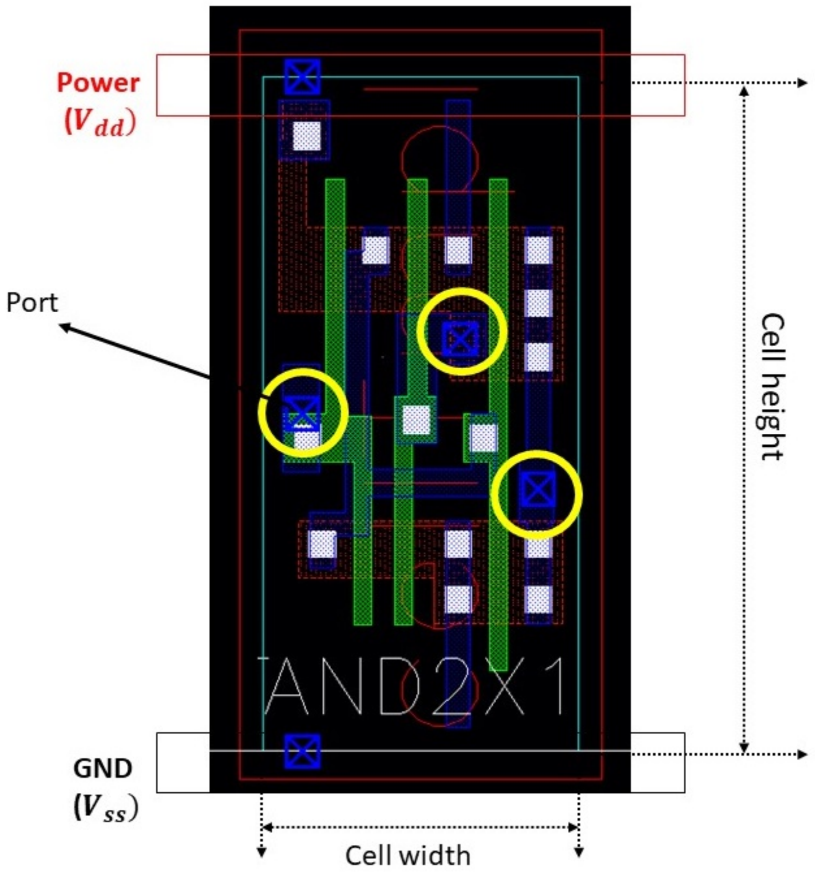


Figure 2.7: 2-input AND cell from GPDK045 library.

The biggest advantage of standard-cell based design flow is that it enables design re-use and shortens the time to market. It makes it possible to massively scale the design while most of the synthesis and implementation details are taken care of by state of the art EDA tools. This also reduces human effort and possibility of errors during the design cycle. The present work is based on the generic 45nm standard-cell library from Cadence (GPDK045). Figure 2.7 illustrates a 2-input AND cell from the the GPDK045 standard-cell library. The height of the standard-cells in a library is maintained as a constant, this makes it possible to align the standard cells in a single row. The width of a cell varies depending on the complexity of the cell, number of transistors, and their drive strength.

2.3. Parasitics Extraction

Parasitics extraction generates a transistor-level netlist which is to used for cell-aware defect modelling. This section provides a brief overview of the operation of parasitics extraction tools. The process of converting the layout view of a design to a transistor-level netlist is known as layout extraction. It serves various purposes - estimating the timing behaviour, noise characterization, power analysis, layout vs schematic comparison, and identifying parasitic elements. Extracting a transistor-level netlist for identifying parasitic elements is known as *parasitic-extraction*. The generated circuit level netlist contains two types of devices - designed devices and parasitic devices. Designed devices are created by the designer, parasitic devices (resistors (R), capacitors (C), and inductors(L)) are a result of material properties. They were not explicitly designed but they exist in the fabricated hardware. As circuits become more complex and interconnect structures shrink further, the effect of parasitic devices on circuit performance becomes an important concern [4]. Most parasitics-extraction tools started out as derivatives of Design Rule Check (DRC) programs, later branching out to evolve as specialized tools [4]. A generic parasitics-extraction flow involves the extraction of designed devices, interconnect and parasitic device extraction (R, L and C), and network simplification tasks as explained in [4]. The behaviour of the extraction tool is customized by the user. These settings not only determine the quality of extraction, but also the suitability of the extracted circuit netlist for the target application.

1. *Determine the nature of extraction:* Identify the target design - analog, digital, mixed signal, Radio Frequency (RF) etc. Zero in on the accepted input database and required output format of the netlist to be generated. Interoperability of EDA tools might also be worth a consideration. Finally, decide whether the extraction would be flat, cell-level or hierarchical in nature. In the flat extraction technique entire design is simplified to a single level of hierarchy. A cell-level extraction system performs parasitic extraction on each individual cell, while only considering the the inter-cell relationships for connectivity. A truly hierarchical system also allows for extracting inter-cell parasitic device. Because of the high sophistication involved in a truly hierarchical system most layout extraction happens at the flat or cell level.

2. *Correlate designed layout dimensions with actual silicon geometries:* A circuit designer builds an ideal view of the different layers of a circuit in terms of polygons which appear as perfect shapes in the software model. The practical limitations posed by lithography and chemical mechanical polishing processes (CMP) result in differences between the layout in a CAD tool vs what is actually achieved. For accurate parasitic estimation an extraction tool must account for these differences. The shape and size of the drawn geometries might be different from what is actually patterned on a silicon substrate. Also the designer has not specified the thickness of the different layers. Empirically these details can be determined by using the available information such as width, spacing between neighbours, density, material properties etc. All of these dimensions are specified in a technology file and provided as input to the extraction tool.

3. *Extraction of designed devices:* The designed device extraction identifies the location of devices such as MOS transistors which have been explicitly designed in the circuit. For example in a typical CMOS design flow a transistor is defined whenever a poly overlaps a diffusion region. A poly-silicon area forms the gate, whereas the drain and source terminals are located in the diffusion regions. An additional step might involve measuring the device parameters such as gate length or width. At this stage the tool might also validate whether the device has been correctly identified or not. For example, a transistor with three source-drain terminals is invalid unless the process architecture allows for such a configuration. The extraction tool recognizes such instances, assigns a unique identifier to each device, makes a note of the terminal connections, device location and parameters and stores this information in a database.

4. *Interconnect or connectivity extraction:* The first step in interconnect extraction is to remove the designed devices, then construct monolithic net structures integrating vias and connections on different layers. Using the information generated in earlier step the device terminals must be located. This helps in assigning unique labels to nets and later to net segments after the parasitic resistance extraction step is complete.

5. *Parasitic device extraction:* After connectivity extraction is complete, each net is broken down into simpler, smaller parts known as segments and the resistance of each sub-part is estimated. The resistance value is estimated using the dimensions(shape), it's nature such as being a linear resistor, contact resistor, bend, junction etc. and a resistance coefficient value defined in the process technology files. Some of the challenges in parasitic R extraction include - treatment of copper interconnects which are not homogeneous, accurate estimation of contact resistors and handling of resistive via-arrays. Specialized techniques have been developed to address these issues and most industrial grade parasitic extraction tool provides options to handle such situations. Capacitance extraction techniques have evolved from 1D, 2D, 2.5D to highly sophisticated 3D field solvers. 1D capacitance extraction uses the area and the perimeter information of the interconnects and routing layers to obtain the capacitance values. In 2D capacitance extraction the lateral or surface capacitance of different layers is also taken into account. In 2.5D extraction two 2D structures are combined to improve the accuracy of parasitic extraction. A 3D

capacitance extractor provides the highest level of sophistication and accuracy. It operates by breaking the cross section into thousands of smaller units and simulating them with field solvers. The generated data is co-related to fit an empirical formula and then estimate the C value between any two segments. Inductance extraction is performed only when required. Since detailed discussion of algorithms involved in parasitic extraction is out of the scope of this thesis, the reader is advised to consult other sources such as [3, 4] for a comprehensive treatment of parasitic extraction techniques.

6. *Circuit reduction:* Parasitic extraction can generate a large number of resistors and capacitors. In most use case scenarios all the extracted R and C elements, internal nodes and net segments might not be required. A simplified version of the netlist obtained by replacing the original R and C elements with the lumped or equivalent values is sufficient. It not only reduces the size of the netlist, but also optimizes the computation requirements of the EDA tools which might operate on these generated netlists.

2.4. QUANTUS QRC BASED PARASITIC EXTRACTION FLOW

Quantus is a parasitics extraction tool offering from Cadence. It supports R,L and C extraction from both cell-level and transistor level designs and is suitable for analog, digital and mixed-signal circuits. Quantus can perform parasitic extraction from multiple process technology nodes, and has built-in support for 3D capacitance extraction using field-solvers. Figure 2.8 illustrates the parasitic extraction flow using Quantus. Quantus can accept various input formats such as DEF, OA (Open Access Database), PVS database (PVS- Physical Verification System) or Calibre (physical verification tool from Mentor Graphics) as inputs. In the Cadence cell-ware flow, PVS database is used as input to generate parasitic extracted netlists in the SPICE format. PVS is a suite of physical verification tools from Cadence which can perform DRC (Design Rule Check), LVL (Layout vs Layout), ERC (Electrical Rule Check), LVS (Layout vs Schematic) and SVS (Schematic vs Schematic). Prior to performing parasitics extraction using Quantus, PVS must be invoked to perform LVS. Firstly, the design flow requires checking the reliability of the layout database. Secondly, PVS generates several input files which are used during RC extraction. For example, Quantus requires information regarding port names, layers, connectivity and device information which is generated by PVS.

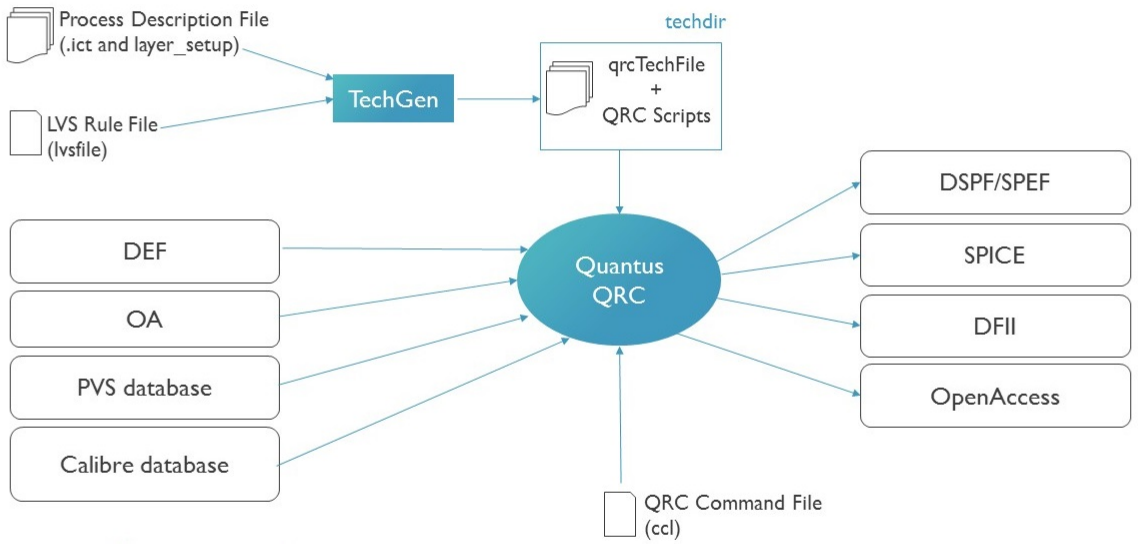


Figure 2.8: Quantus parasitic extraction flow.

Quantus accepts the files produced during PVS to perform RC extraction. The settings for extraction process are specified by the user in the form of an ASCI file known as QRC Command File (.ccl file). The CAD engineer invokes a command line utility known as TechGen to generate the start-up scripts and files containing information about resistance (layer + via) and capacitance (interconnect) models to be used for RC extraction. This is used as an input during the extraction process. Finally, Quantus generates a parasitics-extracted SPICE netlist.

2.5. SUMMARY

This chapter provides an overview of IC testing and it's various building blocks - fault modelling, fault simulation and ATPG. Fault modelling abstracts the actual physical defects in terms of their logical behaviour and therefore enables test pattern generation using ATPG. Fault simulation identifies which test patterns detect which faults. ATPG uses this information to generate test patterns for the entire design. Standard-cell based design flow is most widely used ASIC design methodology. Parasitics extraction generates an accurate simulation model of the circuit by extracting both designed devices and parasitic elements from physical layouts. Quantus QRC is one such parasitic extraction tool which is used in the context of this thesis for generating transistor-level netlists.

3

Cell-Aware Test

Related prior work leading to the evolution of cell-aware test is discussed in Section 3.1. Library characterization and cell-aware ATPG, the two stages of cell-aware test are discussed in Section 3.2 and Section 3.3 respectively. Cell-aware test has been successfully applied for testing complex circuit designs ranging from automotive chips to 32-bit microprocessor designs. Some of these example use cases are reviewed in Section 3.4.

3.1. The Evolution of Cell-Aware Test

Manufacturing tests have largely relied on the use of gate-level fault models such as stuck-at, delay, and bridging fault models [1]. Most of them operate under the assumption that a defect can only take place on the interconnects or I/O ports of gate-level instances [11]. Though this assumption has worked quite well in the past, with the advent of deep sub-micron technologies this is no longer the case [1]. Numerous approaches have been suggested to improve the defect detection capabilities of ATPG-based logic tests. In *n-detect* testing [12, 13] a given stuck-at fault is targeted multiple times by applying *n* different test patterns. New (different) test patterns are generated in the hope that these patterns will improve the defect coverage. This leads to a corresponding increase in the test pattern set making it expensive to apply on large designs. *Embedded Multi-detect* [14] tries a different approach by making use of don't-care bits to detect stuck-at faults. Though, this does not lead to expensive test pattern count increase it prohibits the use of don't care bits for compression and low power testing [15]. The effectiveness of *n-detect* and *EMD* approaches depends on the characteristics of the original test pattern set, which makes the improvement in test quality largely probabilistic [11]. In *gate-exhaustive testing* [16], all possible input test patterns are applied at the library cell-level to guarantee full coverage of all cell-internal defects. For large complex designs having millions of gates, often with multiple inputs this might lead to an explosion of test patterns. Use of SPICE netlists and analog simulations to target transistor level defects, albeit without parasitic elements was proposed in [17]. The cell-aware test methodology evolved as an extension of these ideas and was first introduced in [11]. The objective of cell-aware test is to explicitly target cell-internal defects and improve the defect coverage rather than relying on serendipitous coverage of defects as reported by conventional ATPG [11, 18, 19].This is achieved by modelling realistic physical defects based on actual standard-cell layouts, and simulating them with an exhaustive set of 2^n inputs (where n is the number of cell-inputs). Cell-aware test methodology consists of two stages - library characterization and cell-aware ATPG as illustrated in Figure 3.1. During the library characterization stage the cell-level test patterns which can detect the cell-internal defects are determined. The results are then used during the subsequent cell-aware ATPG stage to generate compact set of test patterns at the chip level. Cell-aware test is expected to provide gate exhaustive test quality at a small increase in test cost. Prior research has already proven the usefulness of cell-aware test over *n-detect* [20], *EMD* [21], and *gate-exhaustive* [22] based approaches. The subsequent sections discuss the library characterization and cell-aware ATPG in greater details.

3.2. Library Characterization

Library characterization stage consists of parasitic extraction, defect location identification, defect injection, and defect simulation as illustrated in Figure 3.1. It starts by extracting transistor-level netlists along with parasitic elements from the physical layouts of standard-cells. Parasitic resistors and capacitors form the

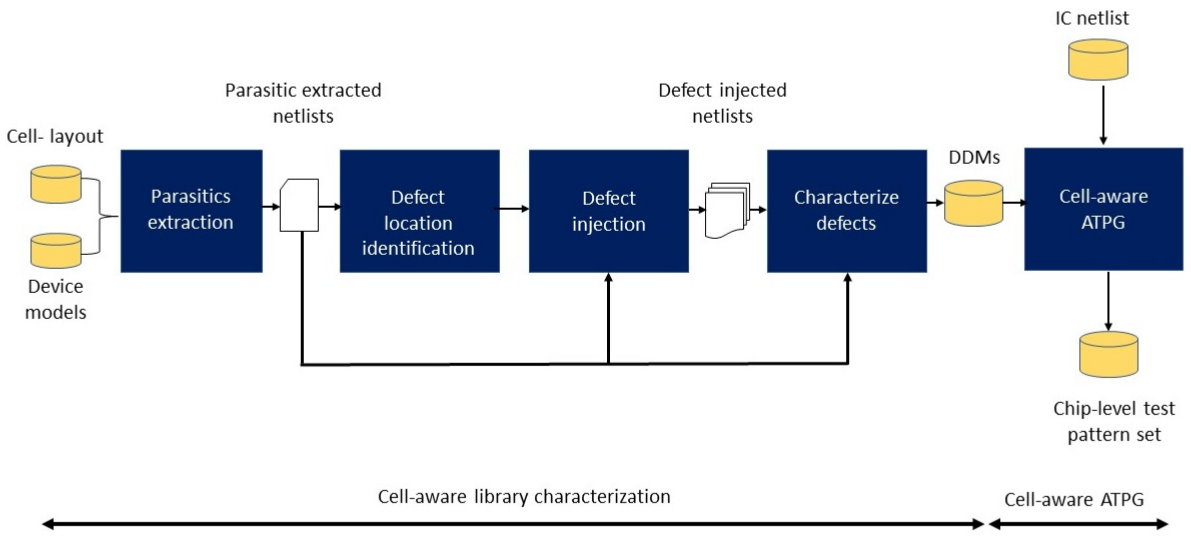


Figure 3.1: Cell-aware test flow.

basis for identifying possible open and short defect sites. A high parasitic resistance (R) is indicative of a long and thin wire and is therefore a potential candidate for an open defect. A large parasitic capacitance (C) between two nets implies they are very close to each other and run in parallel for long distances. This makes them highly susceptible to shorts during manufacturing. Parasitic Rs and Cs above a user defined threshold are considered as target defect candidates. To simulate the defect behaviour, a parasitic resistor is replaced with a hard-open (very high resistance), and a capacitor is replaced with a hard-short (very weak resistance in parallel with the capacitor). Choosing a large threshold value reduces the number of opens and shorts in the target defect set, but might lead to the skipping of some possible defect scenarios. On the other hand, a very low threshold could lead to the modelling of unrealistic defect scenarios. After the completion of the defect extraction step $|D|+1$ netlists per cell are obtained (one defect free and $|D|$ defect injected netlists for $|D|$ defects). Once the defect extraction process is completed, these netlists are subjected to detailed simulations by using an exhaustive set of 2^n patterns, where n represents the number of cell-inputs. This defect characterization process must be repeated for every cell in the library, for all possible defect locations. If the output response is different from the defect free case, then the pattern is considered to detect the defect. The target defect set can be summarized as follows:

- *Candidate open defects:* (1) opens on long and thin wires (2) opens at the transistor source and drain terminals.
- *Candidate short defects:* (1) shorts between two wires which are very close to each other (2) short between transistor drain and source terminals.

Sum	Pattern	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15	d16	d17	d18	d19	d20	d21	d22	d23	d24	d25
9	p0=00/L	0	1	0	0	1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
17	p1=01/L	1	1	1	1	1	0	1	1	1	1	1	0	0	0	1	0	1	1	1	0	0	1	1	0	1
10	p2=10/L	0	1	0	0	1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	1	0	0	0	0
10	p3=11/H	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	1
	Sum	1	3	1	1	3	0	2	4	2	4	4	0	0	0	1	4	4	4	4	0	1	1	1	0	2

Figure 3.2: Defect Detection Matrix (DDM).

The results of this step are then encoded in the form of a Defect Detection Matrix (DDM) which is illustrated in Figure 3.2. A DDM records which cell-level test patterns can detect which cell-internal defect candidates. The number of rows in a DDM are equal to the number of patterns (i.e. 2^n for n-input cell) and each column corresponds to a defect candidate. A 1 indicates that the particular defect d is detected by cell-level test pattern p, 0 means it is not detected. The sum of the rows of the matrix gives the total number of cell-internal defects which can be detected by a particular test pattern. The total number of simulations to be carried out for a cell i in the library is given by:

$$\text{Number of simulations for the } i^{th} \text{ cell } = 2^{n_i} \, . \, d_i \tag{3.1}$$

where n_i is the number of inputs for the i^{th} cell and d_i is the number of defects for cell i. If the entire standard-cell library has c cells, the total number of simulations to be performed are:

$$\text{Total number of simulations for the entire library} = \sum_{i=1}^{c} 2^{n_i} \, . \, d_i \tag{3.2}$$

3.3. CELL-AWARE ATPG

The inputs to the cell-aware ATPG step are the DDMs for all the library cells and the circuit level netlist as illustrated in Figure 3.3. A DDM records which cell-level test patterns can detect which cell-internal defect candidates. This information is used by the cell-aware ATPG engine to identify a compact test pattern set at the circuit-level which can cover the maximum number of cell-internal defect candidates.

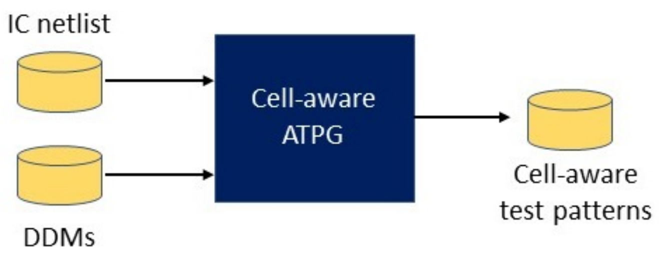


Figure 3.3: Cell-Aware Automatic Test Pattern Generation.

A conventional ATPG engine based on stuck-at faults modelling tries to detect a fault at the cell boundary, i.e. it has to perform a single justification and propagation activity. Cell-aware ATPG on the other hand may have to satisfy several justification targets for a single propagation task. This is best illustrated by an example. Consider a stuck-at 0 fault on the line A0 of the circuit shown in Figure 3.4. The fault effect is justified by making A0 = 1. S0 = 0 and S1 = 0. The other inputs A1 and A2 are don't care bits. The justification and propagation for an example cell-internal defect of the same circuit is shown in Figure 3.5. The required conditions in this case are A2 = 0, A0 = 1, S1 = 0 and S0 = 0. An additional bit A2 must be justified in order to propagate the fault effect. Consequently, compared to stuck-at faults based ATPG, CA-ATPG requires more logic signal values to be justified [19].

3.4. INDUSTRIAL APPLICATIONS OF CELL-AWARE TEST

The cell-aware test methodology has been applied in practice to several multi-million transistor designs laid out in 65, 55, 32 and 28 nm process technology nodes. The results of generating static and transition test patterns for 10 such industrial designs based on the CAT fault model have been reported in [19]. On an average each design had 2.9 million transistors, leading to about 20 million stuck-at faults and 60 million possible CAT defects. The number of potential defect candidates based on CAT model was always significantly higher than static stuck-at faults. The improvement in test quality is attributed to the ability of cell-aware fault model to explicitly target cell-internal defects which otherwise were not detected by conventional fault models. However, this comes at the cost of additional test-patterns. If the improvement in defect coverage is taken into consideration then this increase is marginal. For instance, with 25% increase in test pattern count, the defect coverage for most designs went up by 4%. Production test results of applying CAT to a 32nm notebook processor design from AMD (Advanced Micro Devices) have been presented in [23]. The library characterization

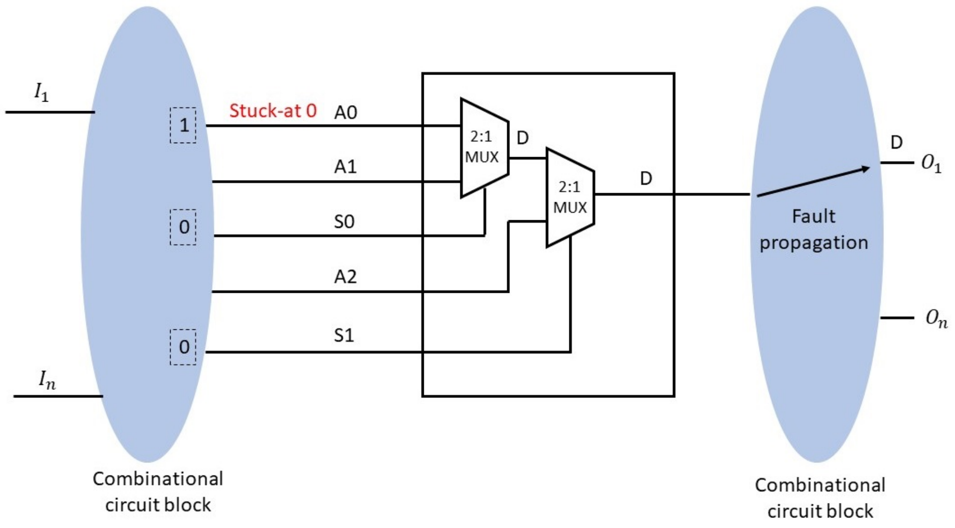


Figure 3.4: ATPG for stuck-at fault model.

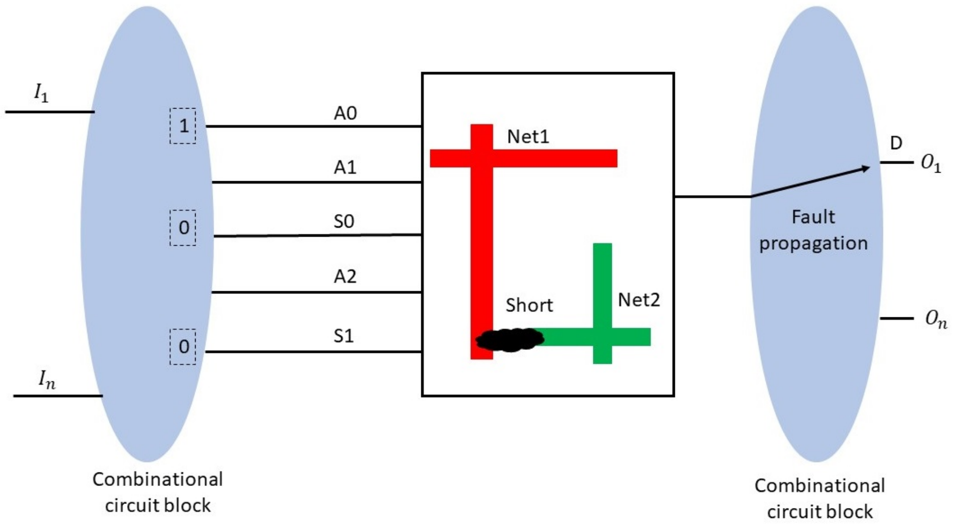


Figure 3.5: ATPG for cell-aware defect model.

was carried out on about 1900 cells. It is reported that an additional 699 failing parts were identified by applying cell-aware patterns which had escaped stuck-at and transition patterns. These statistics were further confirmed by applying system level tests to the failing chips. In [24] production test results of a 130 nm automotive chip from Infineon Technologies based on 216 standard-cell instances are presented. Even on mature technology nodes CAT has effectively improved test quality for mission critical electronic chips which target zero test escapes.

3.5. SUMMARY

Conventional ATPG tools have relied on serendipitous coverage of faults, which can no longer be the case with deep sub-micron technologies. Some of the notable approaches to improving the test quality and increasing the defect coverage include - *n-detect, Embedded Multi-Detect (EMD) and Gate-Exhaustive (GE) testing.* In most cases, either the test application is very expensive, making it highly impractical for actual designs or the improvement in test quality is probabilistic. The cell-aware test methodology has emerged as a promising alternative which uses the actual physical layout as blueprint to identify cell-internal defects. This is done in two stages - cell-aware library characterization and cell-aware ATPG. In cell-aware library characterization a standard-cell library is characterized to generate a set of cell-level test patterns which can detect the targeted cell-internal defects. These are represented in the form of a Defect Detection Matrix (DDM). The cell-aware ATPG stage, uses the DDMs and the circuit netlist to generate a compact set of test patterns for the whole design. Cell-aware test is expected to significantly improve test quality and reduce test escapes at marginal increase in test generation cost. Cell-aware test has been used for the production testing of several multi-million transistor based designs such as microprocessors and automotive chips.

4
ORIGINAL CELL-AWARE FLOW BY CADENCE

Section 4.1 presents an overview of the Cadence cell-aware test flow and it's two stages. Section 4.2 discusses the various shortcomings identified in the library characterization stage and how it affects the overall test quality.

4.1. OVERVIEW OF CADENCE CELL-AWARE TEST FLOW

The Cadence cell-aware test flow is illustrated in Figure 4.1 and is executed in two stages. During the first stage of the cell-aware test flow i.e. the library characterization stage, the cell-level test patterns which can detect the cell-internal defects are determined. The results are stored in the form of a Defect Detection Matrix (DDM), which are then used during the subsequent cell-aware automatic test pattern generation stage to generate a chip level test pattern set.

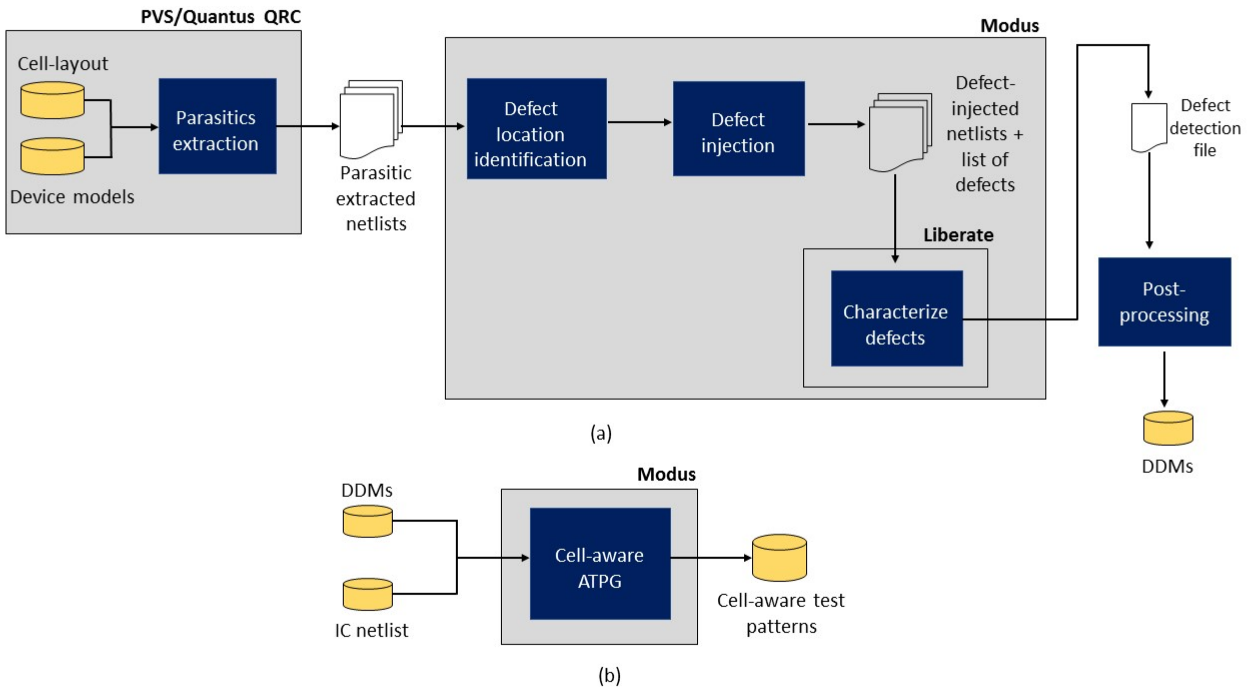


Figure 4.1: Cadence cell-aware (a) library characterization (b) ATPG.

The objective of the library characterization stage is to generate a DDM per library cell. Cadence Quantus QRC is used to generate parasitics extracted transistor-level netlists for all library cells. Using inputs from the technology library, the tool calculates a parasitic resistance for every net segment and a parasitic capacitance between internal nodes. A segment with a large parasitic resistance indicates a long and thin wire which has a higher chances of having an open defect. Parasitic capacitance can be extracted between nodes belonging to different nets (inter-net) or between nodes of the same net (intra-net or self-capacitance). A large capacitance indicates the corresponding nets between which the capacitor is located are close to each other and run in parallel for long distances. Therefore, they have a high chance of being shorted. Based on a user-defined threshold value, a defect location identification script is executed within the Modus software environment which determines the parasitic resistors and capacitors to be considered as opens and shorts in the target defect set. A parasitic resistor above user-defined threshold resistance is modeled as an open by replacing it with a high resistance of 2 $G\Omega$ in the transistor-level netlist. A parasitic capacitor above a threshold is modelled as a short by inserting a low-resistance value of 0.001 Ω in parallel with the capacitor. A high threshold value reduces the number of opens and shorts in the target defect set but might lead to the skipping of some possible defect cases. On the other hand, a very low threshold could mean modelling of unrealistic defect scenarios. In case of transistors, three defects - source-open, drain-open and source-drain short are modeled. Modus generates a separate defect injected netlist for every identified candidate defect. While generating a defect injected netlist the defect extraction scripts inserts place holder values to negate the effect of all other defect candidates other than the one, for which a netlist is being generated. Figure 4.2 illustrates the defect insertion for an open defect on the PMOS transistor source terminal. A hard open of 2 $G\Omega$, is injected on the PMOS source terminal, whereas *potential opens* of 0.001 Ω at the transistor source and drain terminals are inserted at remaining source/drain terminals. A parasitic capacitor is shorted with a high resistance of 2 $G\Omega$ *(potential bridges)*. Similar concept is illustrated for inserting a source-drain short in Figure 4.3. Parasitic resistors and capacitors less than the thresholds are not propagated as defect candidates but remain as a part of the netlist.

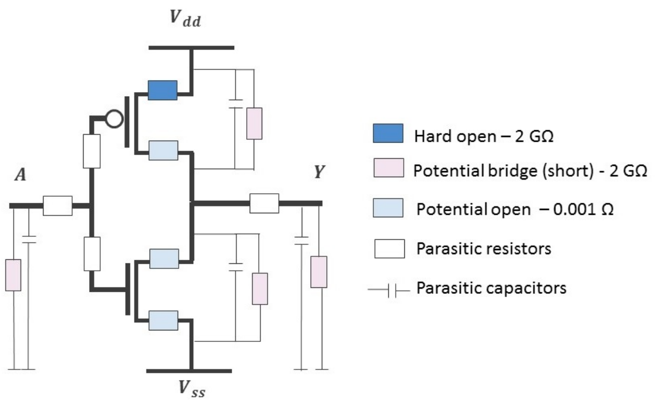


Figure 4.2: Open defect insertion and concept of potential defects.

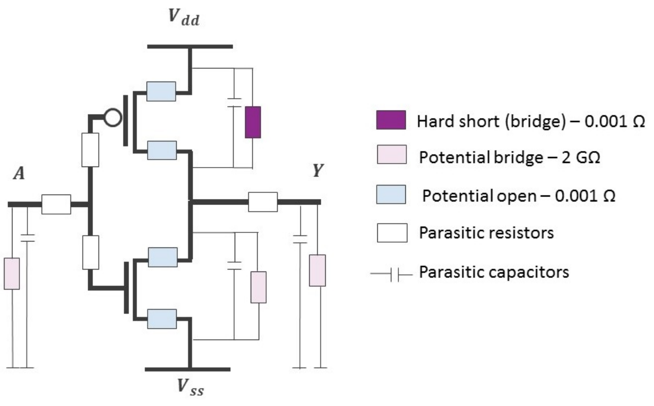
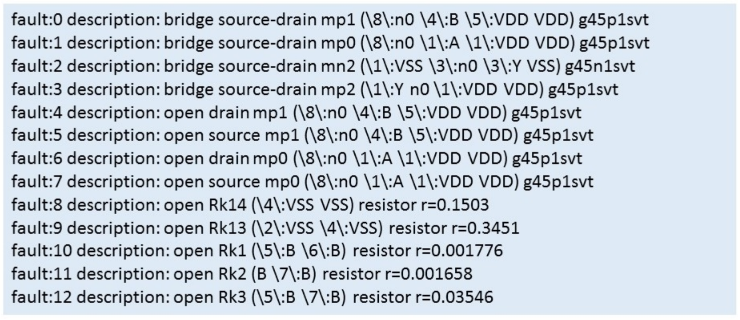


Figure 4.3: Short defect insertion and concept of potential defects.

```
#-----------------------------------------------------------------------------------------------------------------------
# Defect insertion based on potential defects
#-----------------------------------------------------------------------------------------------------------------------

for cell in listofCells do {

        read RES_THRESH, CAP_THRESH;
        read R_POTENTIAL_BRIDGE, R_POTENTIAL_OPEN
        read R_OPEN, R_BRIDGE;

       # parse parasitics-extracted Spectre netlists and store netlist contents
        cellContents = readLines(cell.sp);

        # identify resistors, capacitors and transistor instances in the cell
        mos_primitives = findMosModels(cell, cellContents);
        listofResistor = PATTERN.MATCH('R');
       listofCapacitors = PATTERN.MATCH('C');

        # define lists to hold short, open and transistor defects
        mos_defects = [];
        shorts = [];
        opens = [];

        for capacitor in listofCapacitors
                if capacitor > CAP_THRESH:
                        shorts.append(capacitor = R_BRIDGE);
                        identify capacitor nodes ----- > (n1,n2)
                        bridgeInstance =  R_POTENTIAL_BRIDGE across (n1,n2);
                        bridgeLines.append(bridgeInstance);

        for resistor in listofResistors
                if resistor > RES_THRESH:
                        opens.append(resistor = R_OPEN);

        # identify MOS defects
        for transistor in mos_primitives:
                drain, gate, source, bulk = identifyMosTerminals
                # drain-open defect
                        mos_defects.append(drain = R_OPEN)
                        identify segment ----- > (d1,d2)
                        drainPotentialOpen =  R_POTENTIAL_OPEN between (d1,d2);
                        drainPotentialOpenLines.append(drainPotentialOpen);
                # source-open defect
                        mos_defects.append(source = R_OPEN)
                        identify segment ----- > (s1,s2)
                        sourcePotentialOpen =  R_POTENTIAL_OPEN between (s1,s2);
                        sourcePotentialOpenLines.append(sourcePotentialOpen);
                # source-drain short
                        mos_defects.append(sourcedrain = R_BRIDGE)
                        identify source-drain node pair----- > (sd1,sd2)
                        sourceDrainPotentialBridge =  R_POTENTIAL_BRIDGE between (sd1,sd2);
                        sourceDrainPotentialBridgeLines.append(sourceDrainPotentialBridge);

        for defect in opens:
                write defect injected netlists;
        for defect in shorts:
                write defect injected netlists;
        for defect in mos_defects:
                write defect injected netlists;
```

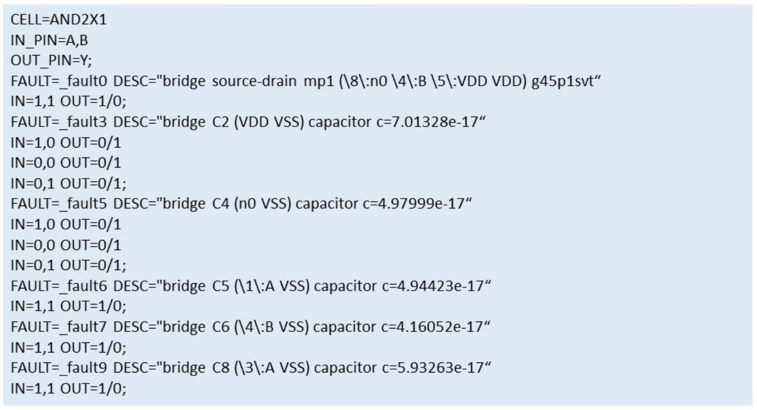
After the completion of the defect extraction step, for D identified defects in the target defect set, $|D|+1$ netlists per cell are obtained (one defect free and $|D|$ defect injected). The list of defects along with a short description are recorded in a faults list file per cell as illustrated in Figure 4.4. This completes the defect extraction step. From within the Modus environment, Liberate is invoked to perform a static simulation on each of these defect injected netlists using an exhaustive set P of cell-level test patterns (where $P = 2^n$ for a cell with n inputs). If the target defect set is D, defect $d \epsilon D$ is considered detected by applying an input pattern $p \epsilon P$ if the output response for this case, differs from the defect-free output. For every defect d, all the cell-level test patterns p which can detect the defect are recorded in a defect detection file as illustrated in Figure 4.5. The defect detection file is generated per-cell and contains the following information:

- fault number and the description of the defect such as defect location and the nature of the defect (open/short).
- input test-patterns which can detect the defect.
- output response of the defect-free and defective netlist.



```
fault:0 description: bridge source-drain mp1 (\8\:n0 \4\:B \5\:VDD VDD) g45p1svt
fault:1 description: bridge source-drain mp0 (\8\:n0 \1\:A \1\:VDD VDD) g45p1svt
fault:2 description: bridge source-drain mn2 (\1\:VSS \3\:n0 \3\:Y VSS) g45n1svt
fault:3 description: bridge source-drain mp2 (\1\:Y n0 \1\:VDD VDD) g45p1svt
fault:4 description: open drain mp1 (\8\:n0 \4\:B \5\:VDD VDD) g45p1svt
fault:5 description: open source mp1 (\8\:n0 \4\:B \5\:VDD VDD) g45p1svt
fault:6 description: open drain mp0 (\8\:n0 \1\:A \1\:VDD VDD) g45p1svt
fault:7 description: open source mp0 (\8\:n0 \1\:A \1\:VDD VDD) g45p1svt
fault:8 description: open Rk14 (\4\:VSS VSS) resistor r=0.1503
fault:9 description: open Rk13 (\2\:VSS \4\:VSS) resistor r=0.3451
fault:10 description: open Rk1 (\5\:B \6\:B) resistor r=0.001776
fault:11 description: open Rk2 (B \7\:B) resistor r=0.001658
fault:12 description: open Rk3 (\5\:B \7\:B) resistor r=0.03546
```

Figure 4.4: An example of a faults list file.



```
CELL=AND2X1
IN_PIN=A,B
OUT_PIN=Y;
FAULT=_fault0 DESC="bridge source-drain mp1 (\8\:n0 \4\:B \5\:VDD VDD) g45p1svt“
IN=1,1 OUT=1/0;
FAULT=_fault3 DESC="bridge C2 (VDD VSS) capacitor c=7.01328e-17“
IN=1,0 OUT=0/1
IN=0,0 OUT=0/1
IN=0,1 OUT=0/1;
FAULT=_fault5 DESC="bridge C4 (n0 VSS) capacitor c=4.97999e-17“
IN=1,0 OUT=0/1
IN=0,0 OUT=0/1
IN=0,1 OUT=0/1;
FAULT=_fault6 DESC="bridge C5 (\1\:A VSS) capacitor c=4.94423e-17“
IN=1,1 OUT=1/0;
FAULT=_fault7 DESC="bridge C6 (\4\:B VSS) capacitor c=4.16052e-17“
IN=1,1 OUT=1/0;
FAULT=_fault9 DESC="bridge C8 (\3\:A VSS) capacitor c=5.93263e-17“
IN=1,1 OUT=1/0;
```

Figure 4.5: Example defect detection file for 2-input AND cell.

In Figure 4.5 fault number 9 is a short between the nodes \1\:A and V_{ss}. This defect is detected by only one input test pattern - 11, the defect-free response is 1 and the faulty response is 0. As the defect-free and faulty outputs are not the same, this defect can be detected. It can be observed that defect number 3 is detected by three test patterns, whereas defect 1 and defect 2 are not detected by any test pattern and therefore not written to the defect detection file. Using the defect detection file and the list of defects generated earlier, a DDM file per cell is derived by running a post processing script. In stage two - cell-aware ATPG, the outputs of the library characterization stage (i.e. DDMs for all cells) along with the gate-level netlist of the entire chip are used to perform ATPG. This generates a compact set of test patterns which covers the maximum possible cell-internal defect candidates for all standard-cell instances in the design.

The Quantus parasitic extraction tool and Physical Verification System (PVS) were already introduced in Chapter 2. Modus is a comprehensive Automatic Test Pattern Generation (ATPG) and silicon-diagnostics tool. From the cell-aware test perspective, the Modus software environment forms the platform for executing the entire cell-aware flow. Liberate is a set of tools for performing cell-level timing characterization, static-analysis, and validation of standard-cells in a technology library. Liberate is invoked within Modus to run a static simulation of transistor-level netlists. Spectre is an analog simulation platform for designing and verifying analog, mixed-signal, and RF circuits. Spectre is used for matching transistor instances in the netlist with their corresponding models available in the technology library, this information is used by Liberate. The list of EDA tools used along with their version information are listed in Table 4.1 for reference.

Table 4.1: Various EDA tools being used in library characterization flow.

Tool	**Version**
Physical Verification System (PVS)	12.10
Quantus QRC	13.10
Modus	17.20
Liberate	17.11
Spectre	17.10

A database containing the layout files, transistor models, software scripts etc. required to execute the flow is available to the users in the form of a Rapid Adoption Kit (RAK). Appendix B provides a brief description of the relevant content from the RAK required for library characterization. Cadence also provides a set of parasitics-extracted transistor-level netlists for all cells in the GPDK045 library. Several issues were identified with the originally provided netlists and these issues are highlighted in Section 4.2. To start the library characterization flow, from the top-level directory execute:

```
modus -f ./SCRIPTS/runmodus.library.cellaware.tcl
```

A detailed description of the software execution flow for *runmodus.library.cellaware.tcl* including background setup tasks and environment initialization is available for reference in Appendix C.

userTime.py - script to measure user CPU time
The *runmodus.library.cellaware.tcl* cannot measure the total time spent in characterizing a particular cell. Operating System (OS) function calls can report on three different timing values - *Real, User and System times. Real time* is the wall clock time. This is the total elapsed time, including the time slices used by other processes and time the process spends in the blocked state, e.g. is waiting for I/O to complete. *User time* is the actual time spent by the CPU when executing the process. *System time* is the amount of CPU time spent in performing system calls within the OS kernel. Therefore, the parameter of interest to us is the user time. This gives the actual CPU time spent in executing our process - i.e. time spent in extracting the faulty netlists and characterizing a particular cell. A top level Python script userTime.py (Appendix A) was created which can read the list of cells to be characterized and automatically logs the user time per cell. This script can be run by executing:

```
python userTime.py
```

4.2. Issues with Cadence Library Characterization Flow

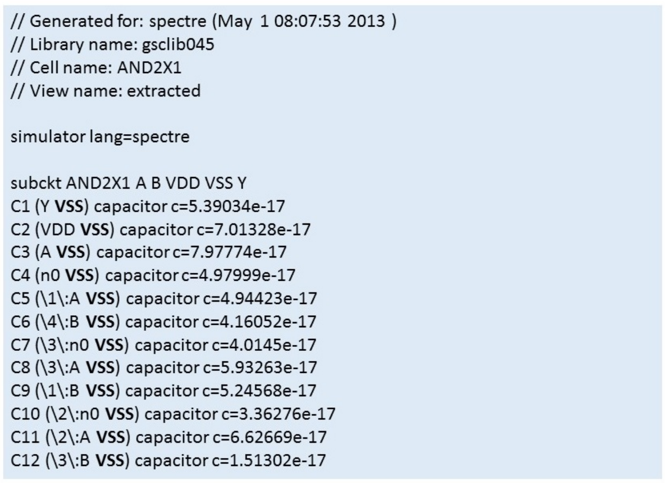
This section discusses the various issues which were identified in the existing library characterization flow.

4.2.1. Parasitic extraction settings

An investigation of the original parasitic extracted netlists provided in the RAK by Cadence revealed several shortcomings. These observation are based on the analysis of parasitic extracted netlists, results of several experiments carried out with different parasitic extraction tool settings, information provided in the tool manuals, and inputs provided by parasitics extraction experts from Cadence. This section highlights the issues, possible solutions to address them are discussed in Chapter 5.

- unfortunately, no information regrading the tool settings used for extracting the originally provided netlists for all cells in the RAK is available.

- parasitic capacitors were always referenced with respect to V_{ss} as illustrated in Figure 4.6. Further investigations revealed that there are two modes of parasitic capacitance extraction - coupled and de-coupled. In the de-coupled mode of parasitic extraction all parasitic capacitors are extracted with respect to a particular reference node. This simplifies the process of parasitic extraction but reduces the accuracy. In the coupled mode, the parasitic capacitors are extracted as is, which increases the accuracy but makes the parasitic extraction process much more complex. In case of the the netlists provided by Cadence, the parasitic capacitors were extracted in de-coupled mode and had to be re-extracted in the coupled mode.



```
// Generated for: spectre (May  1 08:07:53 2013 )
// Library name: gsclib045
// Cell name: AND2X1
// View name: extracted

simulator lang=spectre

subckt AND2X1 A B VDD VSS Y
C1 (Y VSS) capacitor c=5.39034e-17
C2 (VDD VSS) capacitor c=7.01328e-17
C3 (A VSS) capacitor c=7.97774e-17
C4 (n0 VSS) capacitor c=4.97999e-17
C5 (\1\:A VSS) capacitor c=4.94423e-17
C6 (\4\:B VSS) capacitor c=4.16052e-17
C7 (\3\:n0 VSS) capacitor c=4.0145e-17
C8 (\3\:A VSS) capacitor c=5.93263e-17
C9 (\1\:B VSS) capacitor c=5.24568e-17
C10 (\2\:n0 VSS) capacitor c=3.36276e-17
C11 (\2\:A VSS) capacitor c=6.62669e-17
C12 (\3\:B VSS) capacitor c=1.51302e-17
```

Figure 4.6: Original parasitic extracted Spectre netlist.

- most probably no field extractor was utilized for extracting parasitic capacitors and some form of mathematical reduction on RC network was applied. This makes the parasitic extracted netlist to be highly simplified and less accurate for cell-aware defect modelling.

- no information regarding the threshold for parasitic resistance and capacitance extraction used during parasitic extraction is available. The parasitic extraction tool does not extract any resistors or capacitors below this threshold.

- diagnostic information such as the location of the parasitic element, width of the interconnect segment is unavailable in the extracted netlists.

- via resistors are not included, as a result open defects on vias cannot be modeled.

4.2.2. ARBITRARY THRESHOLDS FOR DEFECT INSERTION

Parasitic R's and C's which are below their respective thresholds are not modelled as defects. The threshold values influence the number of candidate defects in the target defect set, thereby affecting the test quality and also the characterization time. In the original flow by Cadence the threshold for R (to be considered as open) was set at 100 Ω and C (to be considered as short) was set at 3 x 10^{-17} Ω. The reasoning behind the choice of these values is not very clear and it seems they were chosen based on a best possible approximation or intuition.

4.2.3. POTENTIAL DEFECTS

In the original cell-aware flow, additional resistors and capacitors which are not actually present in the library-cell but superfluous elements to reserve a possible defect location are introduced into the netlist. These elements known as potential defects, not only change the characteristics of the netlist but also result in inaccurate defect simulations. They must be eliminated to improve the quality of defect simulation and the speed of execution.

4.2.4. RESISTANCE VALUES FOR MODELLING OPENS AND SHORTS

A parasitic resistor above threshold is modelled as an open defect by replacing it with a hard-open resistance of 2 $G\Omega$, and a short is modelled by using a hard-short of 0.001Ω. The defect inserted values can be made even more harder to further improve the test quality and ensure the defect behaviour is captured as accurately as possible.

4.2.5. SIMULATION OF $|D|$ + *1* DEFECT INJECTED NETLISTS

Based on the user-defined threshold values for parasitic resistors and capacitors a candidate set of defects is identified. For D identified candidate defects per cell, $D+1$ netlists (N defect injected netlists and 1 defect-free) are generated. Liberate, the tool being used for defect simulation must perform some background tasks every time it encounters a new netlist. These tasks might include - reading the netlist, identifying the cell parameters, location of the I/O pins etc. For individual cells the hardware resources and the time required to complete these background tasks might not be very significant. But when we consider the sum total of the time spent for all the faulty netlists this becomes a significant value. This can be accomplished more efficiently by exploiting certain in-built features of Liberate as discussed in Chapter 5.

4.2.6. SPECTRE NETLIST FORMAT

One more important issue identified during the course of this project was regarding the format of the parasitic extracted netlists. If the Cadence CAT flow can only accept Spectre format, this might discourage potential customers from adopting this flow. Especially if it requires them to invest in new tools just for generating input netlists in a specific format. Even if they decide to invest in such tools, additional steps are required for generating netlists in Spectre format as illustrated in Figure 4.7. This might be unnecessary and a waste of time and efforts. For example, for a relatively small subset of standard-cells (313 cells) this translates to 6 hours of additional compute time. This increases proportionately when the number of cells being considered goes up. Because of these reasons it is advisable to modify the Cadence CAT flow to accept more generic and widely accepted formats such as SPICE. *(*ADE (Analog Design Environment) is an analog design and verification toolkit.*

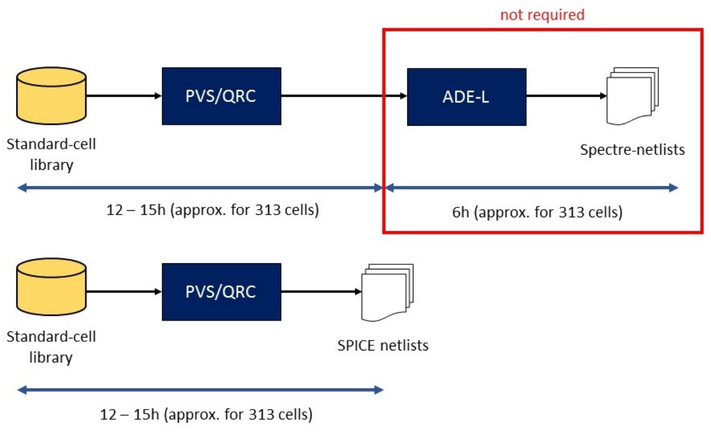
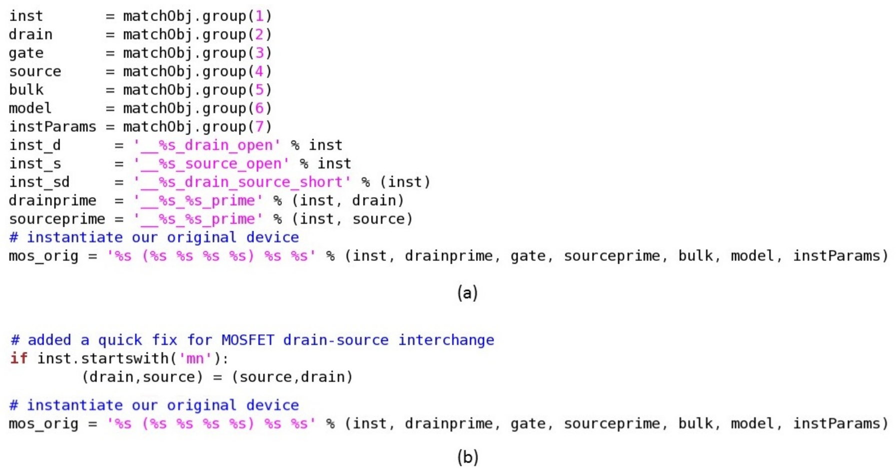


Figure 4.7: Additional steps required for generating Spectre netlists.

4.2.7. REVERSAL OF TRANSISTOR SOURCE-DRAIN TERMINALS

Conceptually, a transistor is symmetric with respect to its drain and source terminals. The drain and source terminals are decided based on the voltage levels at the two terminals. During defect extraction, the script reads the extracted Spectre netlists and matches the transistor instances with transistor models available in the technology library. For every transistor three defects (source-open, drain-open and source-drain short) are inserted - and a new netlist is generated per defect. If an open is being inserted on the source terminal, the corresponding segment must be split and a resistor is inserted in the gap to model a source-open defect. It was observed that during this process of defect insertion on the transistor terminals, a software bug caused the drain and source terminals to be reversed. This issue and the software fix added are illustrated in Figure 4.8. Determining the drain and source terminals is important for correctly identifying the location of the the defect during later diagnosis stages. This avoids the possibility of treating an open on the source as a drain-open or the other way round.



```
inst       = matchObj.group(1)
drain      = matchObj.group(2)
gate       = matchObj.group(3)
source     = matchObj.group(4)
bulk       = matchObj.group(5)
model      = matchObj.group(6)
instParams = matchObj.group(7)
inst_d      = '__%s_drain_open' % inst
inst_s      = '__%s_source_open' % inst
inst_sd     = '__%s_drain_source_short' % (inst)
drainprime  = '__%s_%s_prime' % (inst, drain)
sourceprime = '__%s_%s_prime' % (inst, source)
# instantiate our original device
mos_orig = '%s (%s %s %s %s) %s %s' % (inst, drainprime, gate, sourceprime, bulk, model, instParams)
```

(a)

```
# added a quick fix for MOSFET drain-source interchange
if inst.startswith('mn'):
        (drain,source) = (source,drain)

# instantiate our original device
mos_orig = '%s (%s %s %s %s) %s %s' % (inst, drainprime, gate, sourceprime, bulk, model, instParams)
```

(b)

Figure 4.8: (a) Software bug causing reversal of transistor source-drain terminals (b) fix added.

4.3. SUMMARY

This chapter provided an overview of the Cadence CAT flow with specific details about the library characterization. The flow is based on several EDA tools - Quantus/PVS, Modus, Liberate, and Spectre. Software scripts are executed within Modus for identifying open/short defects, creating defect injected netlists, simulating them with cell-level test patterns, and finally writing out the simulation results to a defects detection file. The inputs to the library characterization stage are the cell-layouts and the technology files such as transistor models; the outputs are the defect-injected netlists, faults list file and the results of characterizing the defective netlists using static simulation. The final results are encoded in the form a matrix and written to a DDM file per cell. The DDMs along with the netlist of the complete circuit design are then used for generating circuit-level test patterns during the CA-ATPG stage. The original parasitic-extracted netlists provided as a part of Cadence RAK for CAT suffer from variety of problems which need to be fixed to improve the quality of Cadence library characterization flow. CAT is a non-standard application of parasitic extraction, therefore it is essential to define a set of customized settings for the parasitic extraction tool to generate transistor-level netlists which are well-suited for cell-aware defect modelling.

5
IMPROVEMENTS TO THE CELL-AWARE TEST FLOW

Section 5.1 starts by giving an overview of the related prior work on cell-aware test and approaches to improving the cell-aware flow. Section 5.2 talks about the customized settings to be used for the Quantus parasitic extraction tool for extracting transistor-level netlists targeting cell-aware defect-modelling. Section 5.3 discusses regarding the threshold values for parasitic resistors and capacitors for considering them as defect candidates. Section 5.4 discusses regarding the drawbacks of potential defects, which were superfluous elements being introduced into the netlist and the improvements possible by eliminating them. Section 5.5 talks about using super hard defect values for modelling opens and shorts. Section 5.6 discusses the speedup achieved by concatenating all defective netlists together as a part of a single large netlist prior to simulation. Finally, Section 5.7 talks about the reduction in the number of defect candidates by inserting a single short per net pair.

5.1. RELATED PRIOR WORK

It is being increasingly recognized that for newer and smaller technology nodes, a significant number of manufacturing defects are occurring within the standard-cells [1, 11]. Test-pattern generation based on traditional stuck-at fault model will not be sufficient, this is especially true for critical electronic systems such as automotive and medical which cannot tolerate any defective chips. The cell-aware test methodology was first introduced in [11] by explicitly targeting low resistive cell-internal short defects (1Ω). This paper showed that by applying cell-aware test patterns an improvement of upto 1.2% in defect coverage was possible for the target library over pure stuck-at test patterns. In [18] this was enhanced to cover hard opens (1 $G\Omega$) and weaker short defect candidates. All of this work was consolidated, apart from the inclusion of small delay defects in [19]. Cell-aware test for sequential cells such as scan flip-flops was introduced in [25, 26]. Several use case scenarios of applying cell-aware test patterns for the production testing of designs ranging from notebook processors [23] to high-quality automotive chips [24] are available in the literature. Recently, the usefulness of cell-aware ATPG for programmable chips (FPGA's) targeting the automotive segment was presented [27]. As cell-aware test is increasingly being adopted by semiconductor companies, optimizing the library characterization process for improving the quality of defect simulation and reducing the computation time is a topic of great interest and ongoing research.

The effectiveness of the cell-aware test approach depends heavily on the quality of the library characterization outputs. Most prior work does not seem to disclose the complete details about how intra-cell defects are modelled during library characterization stage. For example, in [19] and most of the earlier work on cell-aware test leading to this paper, no information about how exactly transistor-level netlists are extracted from standard-cell layouts for cell-aware defect modelling is available. In [28] the authors propose an algorithm to combine the cell layout information along with the parasitic-extracted netlists during the defect injection process to reduce the target defect set, and hence improve defect simulation time. The process is executed in two steps: layout identification and defect injection. During the layout identification stage the cell-layout is converted from a binary format to a plain-text format using an open source tool *gds2gdt*. An algorithm identifies the rectangular polygons which are situated on the same layer and breaks them into individual blocks or segments. Transistor terminals are identified, followed by locating the vias. Rectangular segments and vias which are connected to each other and form a continuous structure belong to the same net and are grouped together. Once this is done, the distance between two adjacent net pairs on the same layer is calculated. If this distance is less than the user defined threshold, then the net pair is added to the short defect list. A tracing algorithm records one open defect per all net segments of the same net located on the same routing layer. The list of open and short defects are used in the subsequent fault injection stage to determine the exact location and then insert an open or short defect in the parasitic- extracted netlist. Even though this algorithm is very useful and the authors have reported a reduction of upto 80% in number of open and short defects to be considered for cell-aware library characterization, there are some open questions. Firstly, the proposed approach doesn't seem to do anything radically different - it still uses parasitic extraction and additionally requires a post processing step. Secondly, no information is available on how the threshold values for injecting shorts between nets is decided. And finally, this paper is based on the assumption that inter-layer shorts are not possible and hence excludes them. Which is not always the case, especially for smaller technology nodes as discussed in [19]. Instead of running a post-processing script which takes up additional computation time the above mentioned features can be directly implemented in the defect extraction script, and by appropriately choosing the settings in the parasitics extraction tool itself. This is one of the objectives of this thesis work.

5.2. PARASITIC EXTRACTION SETTINGS FOR CELL-AWARE TEST

Parasitics extraction is used for generating an accurate simulation model of the circuit. A parasitics extraction tool uses the layout information along with the technology library files to estimate the parasitic resistance, capacitance, and inductance. The output of a parasitic- extraction process is a transistor-level netlist, which can be used for various purposes such as signal integrity analysis and delay estimation. Cell-aware test is a non-standard application of parasitic-extraction. For example, estimation of signal delay often involves applying some form of circuit simplification to reduce the number of parasitic resistors and capacitors in the extracted netlist. This might not be suitable for cell-aware test. Also, most applications of parasitics-extraction take into account the extracted inductance parameter, which might not be necessary for cell-aware test. The objective of cell-aware test is to use the parasitic-extracted transistor level netlists to model realistic defects, and identify cell-level test patterns which can detect these defects. These cell-internal defect candidates must include opens on transistor source, drain and gate terminals; as well as on interconnects on same routing layer and between layers. Shorts must be detected between transistor terminals and between interconnects within the same layer or adjacent layers. Since parasitic extracted resistors and capacitors form the basis for modelling cell-internal defects, the quality improvement possible because of cell-aware test depends heavily on the input transistor-level netlists. Therefore, the parasitic-extraction tool must be tuned to extract transistor-level netlists with parasitic resistors and capacitors which allow us to model defects at the above mentioned defect sites. These settings are organized under various sub-groups and given as inputs to the extraction tool in the form of a command control file (ccl). The process of generating, and executing parasitic extraction using the ccl file was discussed in Chapter 2. The general settings for Quantus QRC tool are specified as follows:

- *input_db -type pvs*: The input database format to be read by Quantus comes from PVS outputs (discussed in Chapter 2).
- *extract -type c_coupled*: extract all possible parasitic resistors and capacitors from all nets, perform capacitance extraction to extract the capacitors without coupling them with a specific reference node. This setting addresses the issue discussed in Section 4.2.1 which is illustrated in Figure 4.6.
- *output_db -add_explicit_vias true*: Vias are also susceptible to manufacturing defects, by selecting this option the Quantus parasitic extraction tool inserts a parasitic resistor for all via locations and includes them in the netlist. This ensures such defects can also be modeled during the library characterization stage.
- *output_db -spice*: generate SPICE netlists to address the issue discussed in Section 4.2.6.
- *output_db -include_parasitic_res_model 'comment'*
  output_db -include_parasitic_res_width 'drawn'
  output_db -output_xy 'PARASITIC_RES' 'PARASITIC_CAP'

Including comments such as location of parasitic R's and C's, layer information, *x,y coordinates* and segment widths, helps during diagnosis.

The parasitic extraction tool must be configured such that open defects on transistor terminals, and on interconnects within a layer as well as between layers are identified. The Quantus tool must extract all relevant parasitic resistors, and the associated settings are outlined below:

- *extraction_setup -max_fracture_length infinite*: this ensures that segments are not partitioned into smaller parts and treated as one monolithic unit. An open defect on an interconnect segment exhibits the same behavior whether it is located to the right, left or in the middle. So a continuous segment must not be decomposed into smaller units. This also reduces the number of parasitic resistors and avoids the unnecessary simulations of defects which are equivalent.
- *filter_res -merge_parallel_res true*: standard-cell libraries are not expected to contain parallel-segments.
- *filter_res -merge_parallel_via*
  filter_res -max_via_array_size 'auto':

  By explicitly including a parasitic resistor for every via it is possible to model open defects on vias. However, the cell-aware defect insertion model operates on the principle of 'one-defect at a time'. Therefore parallel vias need to be merged, and a single equivalent open defect is modelled for the resulting resistance.
- *filter_res -remove_dangling_res true*: parasitic resistors from dangling segments do not cause any faulty behaviour, and hence do not affect the simulation accuracy. But dangling segments can result in shorts, by choosing this option the dangling resistors are not listed in the netlist, but the capacitors attached to them are still included in the netlist.
- *filter_res -min_res 1e-10*: This threshold value i.e. *MinR* determines the cut-off criteria for extracting parasitic resistors. This value must be chosen such that all possible parasitic resistors are included in the extracted netlist. In the original netlists provided by Cadence, no information regarding this value was available. For determining the appropriate value for *MinR* multiple parasitic-extractions were performed by setting extremely low values for *MinR*. This was an exercise to estimate the lowest possible range of magnitudes for extracted parasitic resistors. It was noticed that for thresholds less than 1 x 10^{-10} Ω no significant variation in the quantity or characteristics of the resistors extracted was observed. Therefore, in the modified flow *MinR* is set at 1 x 10^{-10} Ω.

A large capacitance means two nets are very close to each other, run in parallel for longer distances, and therefore can be easily shorted. The cell-aware flow must identify shorts between terminals of a transistor, shorts within the same layer as well as between layers which are adjacent to each other. The appropriate settings for the Quantus parasitic extraction tool to achieve this are:

- *filter_cap -exclude_self_cap -true*: Self-coupling capacitors between nodes on the same net are excluded. If these capacitors are included, this would lead to irrelevant shorts of a net with itself.
- *filter_cap -exclude_floating_nets true*: floating nets are rare in highly optimized library-cells, even if they occur this option removes them. Shorts with floating nets do not cause any faulty behaviour.
- *filter_coupling_cap*
  -coupling_cap_threshold_absolute 1e-25
  -coupling_cap_threshold_relative 1e-3
  This threshold value i.e. *MinC* determines the cut-off criteria for filtering parasitic capacitors. This value must be chosen such that parasitic capacitors between two nodes which are far apart are eliminated. For determining the appropriate value for *MinC* multiple parasitic-extractions were performed by setting extremely low values for *MinC*. This was an exercise to estimate the lowest possible range of magnitudes for extracted parasitic resistors. It was noticed that for thresholds less than 1 x 10^{-25} F no significant variation in the quantity or characteristics of the resistors extracted was observed. Therefore, in the modified flow *MinC* is set at 1 x 10^{-25} F.
- *use_field_solver high_accuracy*
  field_solver_type probabilistic

```
#-------------------------------------------------------------------------------------------------------------------
# Parasitics-extraction to generate netlists for cell-aware library characterization.
#-------------------------------------------------------------------------------------------------------------------

for all cells ε listofCells do RC_Extraction {

    # initialize software environment
     initialize Quantus QRC environment;
     set path to read technology library;
     set working_directory path;

     # set technology library
     set process_technology_lib = 'GPDK045'

     # set input database for parasitics extraction
     set input_db -type = 'PVS database'

     # set output netlist type
     set output_db_type = 'spice'

     # include diagnostics information in the netlist
    -include_cap_model "comment" \
     -include_parasitic_cap_model "comment" \
     -include_parasitic_res_model "comment" \
     -include_parasitic_res_width_drawn  = true \
     -output_xy  coordinates for R, C and transistor devices

     # set options for C extraction and use 3D field solver for C extraction
     set extract \
      -   selection "all" \
      -  type "c_coupled" \
      -  use_field_solver "high_accuracy" \
      -  field_solver_type "probabilistic"
      set exclude_self_cap = true

    set  filter_coupling_cap \
      - coupling_cap_threshold_absolute 1e-25 \
      - coupling_cap_threshold_relative 0.001
      - exclude_floating_nets = 'true'

     # set options for resistance extraction
     set merge_parallel_res  = true
     set min_res  = 1e-10
     set remove_dangling_res  = true
     set -max_fracture_length  = 'infinite'
     set remove_dangling_res = 'true'

     # set options for vias
     set add_explicit_vias  = 'true'
     set filter_res -merge_parallel_via = true
     set filter_res -max_via_array_size  = 'auto'

     }
```

Quantus parasitics extraction model
Based on the above mentioned settings Quantus parasitics extraction tool extracts transistor-level netlists which are well-suited for cell-aware defect modelling. The Quantus PEX tool divides a *net* that electrically connects two or more *terminals* into net *segments*, which are bounded by terminals or *internal nodes*. A net is divided into multiple net-segments if a *fork* or a transition from one layer to the other or a sharp corner bend is encountered. Segments can be within the same layer (intra-layer) or inter-layer (contacts, vias). This is illustrated in Figure 5.1 and is based on the following definitions:

1. *Terminals:* Cell input/output pins, transistor terminals (source, drain, gate or bulk), V_{dd}/V_{ss}.
2. *Net:* set of electrically connected terminals.
3. *Segment:* part of net with individual parasitic segments.
4. *Node:* end point of a segment.

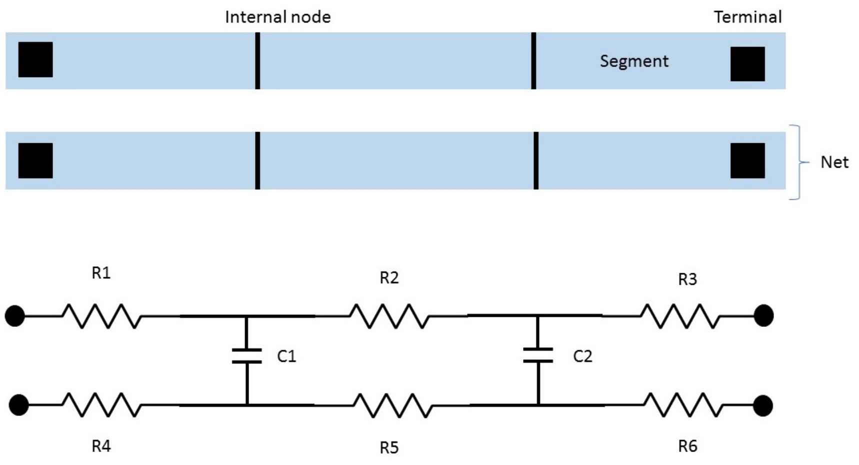


Figure 5.1: Terminals, segments, nets and nodes.

Figure 5.2 illustrates the parasitics-extraction result for an inverter (INVX1) cell from the GPDK045 library. The cell-library has four layers: *n-diffusion*, *p-diffusion*, *poly-silicon*, and *metal-1*. Contacts exist from metal-1 to other layers. The inverter cell has one input and one output terminals and two transistors. The Quantus PEX tool has also identified various internal nodes as illustrated in the figure. A script for automating the Quantus parasitics extraction flow is presented in Appendix A.

5.3. THRESHOLD VALUES FOR DEFECT INSERTION

For cell-aware library characterization two user-defined threshold values are used, R_{th} and C_{th}. Any parasitic resistor below R_{th} will not be modeled as an open, and a parasitic capacitor below C_{th} will not be modeled as a short. Threshold values must be defined such that all realistic defect locations are covered and unlikely defects are neglected, this avoids futile defect simulations and saves valuable simulation time. During resistance extraction the parasitics extraction tool divides a net into multiple segments whenever it encounters an inter-layer transition or fork. Each of these segments represents a parasitic resistor and is therefore a possible location for an open defect. In the original cell-aware flow, the value for R_{th} was set at 100 Ω. That is, parasitic resistors below 100 Ω were not considered as open defect candidates. In the modified flow, the resistance threshold R_{th} is being changed to 0.0 Ω to consider all parasitic resistors as open defect locations. By setting a zero threshold for parasitic resistors all possible open-defect locations, including possible weak-opens are covered. The magnitude of the parasitic capacitance is indicative of the possibility of a short between two nets. A low capacitance means the two nets are far apart in the layout, and a short between them is highly unlikely. Two nets which are very close, run in parallel for long distances and therefore have a high capacitance and can be easily shorted. In the original cell-aware flow, the threshold for short defect insertion was set as $3 \text{ x } 10^{-17}$ F, the choice of this threshold seemed arbitrary, the reasons behind choosing this threshold were

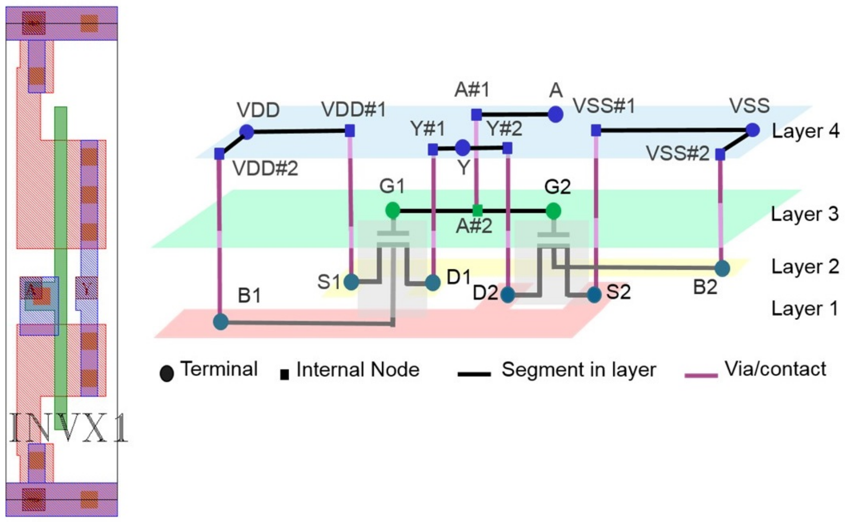
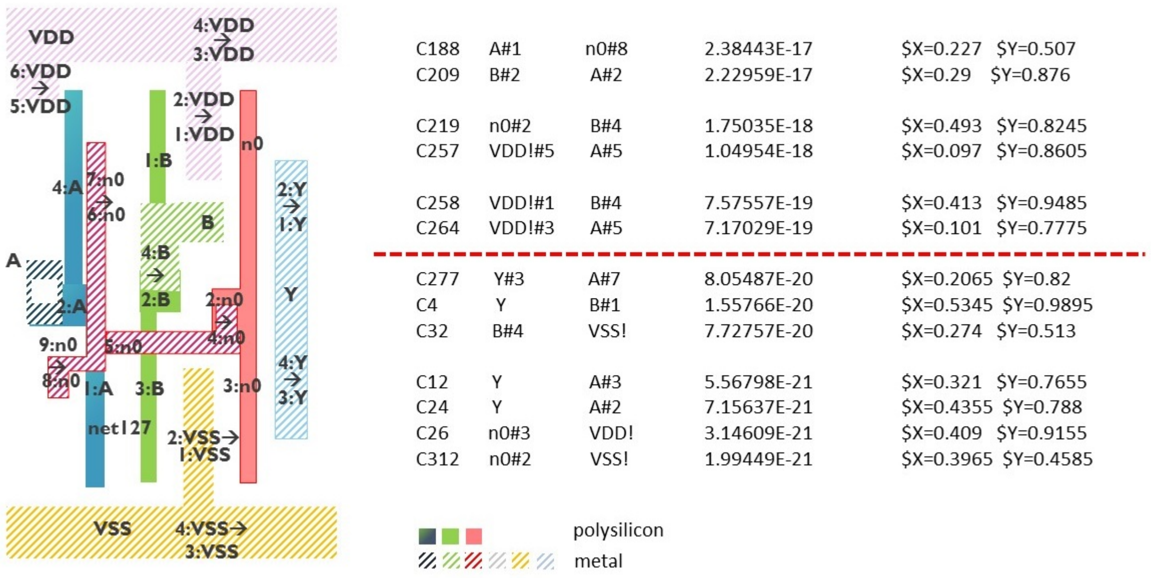


Figure 5.2: Quantus parasitics-extracted cell model.

unknown. A heuristics-based approach for choosing a suitable value for C_{th} is proposed. By correlating the magnitudes of the parasitic capacitors, with the locations of the nets between which they are located, a suitable value for C_{th} can be determined. Consider the annotated layout for the two input AND cell (AND2X1) from the GPDK045 library illustrated in Figure 5.3. The two nets A and Y are relatively far from each other, consequently the value of the parasitic capacitance between Y and *A:3* (segments of net Y and A) is 5.56 x 10^{-21} F. On the other hand, the nets *n0* and *B* are very close to each other and the parasitic capacitance extracted between the two nets is of the order of 10^{-18} F. Therefore by mapping the magnitude of the parasitic capacitance with the location of nets between which it is located, a rough estimate of the possibility of a short between the two nets can be predicted. Based on these approximations, the C_{th} for this library is set at 1 x 10^{-19} F. Below this threshold it is highly unlikely that a short might occur. This approach was tried on two small cells in the GPDK045 library, a two input AND and an inverter (INVX1). Standard-cell layouts a have uniform height but vary in widths, they share common design principles. If this approximation holds true for the smaller cells such as inverters, then it must be true for all other cells in the library.



C188	A#1	n0#8	2.38443E-17	$X=0.227 $Y=0.507
C209	B#2	A#2	2.22959E-17	$X=0.29 $Y=0.876
C219	n0#2	B#4	1.75035E-18	$X=0.493 $Y=0.8245
C257	VDD!#5	A#5	1.04954E-18	$X=0.097 $Y=0.8605
C258	VDD!#1	B#4	7.57557E-19	$X=0.413 $Y=0.9485
C264	VDD!#3	A#5	7.17029E-19	$X=0.101 $Y=0.7775
C277	Y#3	A#7	8.05487E-20	$X=0.2065 $Y=0.82
C4	Y	B#1	1.55766E-20	$X=0.5345 $Y=0.9895
C32	B#4	VSS!	7.72757E-20	$X=0.274 $Y=0.513
C12	Y	A#3	5.56798E-21	$X=0.321 $Y=0.7655
C24	Y	A#2	7.15637E-21	$X=0.4355 $Y=0.788
C26	n0#3	VDD!	3.14609E-21	$X=0.409 $Y=0.9155
C312	n0#2	VSS!	1.99449E-21	$X=0.3965 $Y=0.4585

Figure 5.3: AND2X1 layout to illustrate heuristics for choosing C_{th}

5.4. Elimination of Potential Defects

As discussed in Chapter 4 the original cell-aware flow by Cadence inserted additional resistors and capacitors known as *potential defects* in the netlist. These resistors and capacitors were not a part of the parasitic extracted netlist, but were superfluous elements to act as place holder values for possible defect sites. The procedure for generating defect-injected netlists is as follows. The defect-insertion script starts by parsing the defect-free parasitic-extracted netlist. The transistor instances and their pin terminals are identified by matching their definitions with the SPICE models available in the technology library. For every transistor, three defect injected netlists are generated to cover source-open, drain-open, and source-drain short defects. Source-open and drain-open defects are modeled by splitting the segments connecting the source/drain transistor terminals and inserting a hard open. A source-drain short is modeled by inserting a hard-short between the source and drain terminals. Apart from the defect for which a defect injected netlist is being generated all other possible open defect locations on transitor terminals are replaced with *potential opens*, whereas *potential shorts* are placed in parallel with parasitic capacitors. Figure 4.2 illustrates this defect insertion procedure for a source-open defect on the PMOS transistor of an inverter circuit. The inclusion of additional resistors and capacitors completely changes the characteristics of the circuit causing inaccurate defect simulations. To eliminate 'potential defects' the defect insertion script was modified such that only a single defect is modeled per defect-injected netlist, and no additional resistors or capacitors are inserted. Figure 5.4 illustrates the defect insertion process before and after eliminating potential defects.

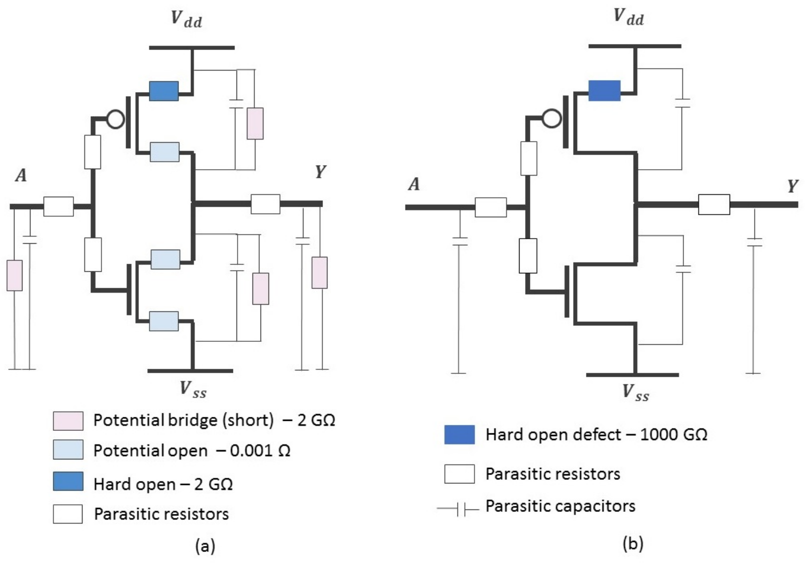


Figure 5.4: Opens (a) based on potential defects (b) without potential defects.

5.5. Insertion of Super-Hard Defects

Parasitic resistors and capacitors in a transitor-level netlist form the basis for identifying possible locations for open and short defects. A parasitic resistor is replaced with a a very high resistance to model an open, and an extremely low resistance is inserted in parallel with a parasitic capacitor to model a short. For an open defect, higher the value of this defect resistance better is its ability to model an actual open. Similarly, for a short lower the value of the defect resistance better is it is ability to accurately model an actual defect. For ATPG a hard defect has much bigger impact and is usually much easier to be detected by a cell-level test pattern. A weaker variant of the defect might go undetected during defect simulation. By inserting super hard defects additional cell-level test patterns might be to able detect this defect. This is beneficial during ATPG, as the ATPG tool now has a wider pool of of cell-level test patterns to choose from.

```
#-----------------------------------------------------------------------------------------------------------------------
# Defect insertion without potential defects
#-----------------------------------------------------------------------------------------------------------------------

for cell in listofCells do {

        read RES_THRESH, CAP_THRESH;
        read R_POTENTIAL_BRIDGE, R_POTENTIAL_OPEN
        read R_OPEN, R_BRIDGE;

       # parse parasitics-extracted Spectre netlists and store netlist contents
        cellContents = readLines(cell.sp);

        # identify resistors, capacitors and transistor instances in the cell
        mos_primitives = findMosModels(cell, cellContents);
        listofResistor = PATTERN.MATCH('R');
       listofCapacitors = PATTERN.MATCH('C');

        # define lists to hold short, open and transistor defects
        mos_defects = [];
        shorts = [];
        opens = [];

        for capacitor in listofCapacitors
                if capacitor > CAP_THRESH:
                        shorts.append(capacitor = R_BRIDGE);

        for resistor in listofResistors
                if resistor > RES_THRESH:
                        opens.append(resistor = R_OPEN);

        # identify MOS defects
        for transistor in mos_primitives:
                drain, gate, source, bulk = identifyMosTerminals
                # drain-open defect
                        mos_defects.append(drain = R_OPEN)
                # source-open defect
                        mos_defects.append(source = R_OPEN)
                # source-drain short
                        mos_defects.append(sourcedrain = R_BRIDGE)

        for defect in opens:
                write defect injected netlists;
        for defect in shorts:
                write defect injected netlists;
        for defect in mos_defects:
                write defect injected netlists;
```

5.6. Reduction in Run-Time by Modifying Software Flow

In the original cell-aware flow, for D defects in a cell, $|D| + 1$ netlists per cell were simulated (one defect free and $|D|$ defect injected). One of the drawbacks of this approach is that, it involves a lot of boilerplate code execution which can be avoided. For example, rather than initializing the Liberate simulation environment every time a defective netlist is encountered it can be done only once. The software scripts for invoking Liberate are modified to process all the defect-injected netlists as part of one single loop. This change also requires concatenating the defect-injected netlists one after the other to create a single large netlist file. Consequently, the defect-injected netlists become independent sub circuits within a larger netlist file, rather than existing as separate netlists in the memory. The repeated execution of background setup tasks is avoided reducing the simulation time, this is depicted in Figure 5.5. For initial prototyping and verification purposes, the single large netlist file had to be generated by hand and the Liberate software scripts were modified to accommodate this change. It was not practical to repeat these steps on all the 313 parasitic-extracted transistor netlists, as the manual editing was prone to errors and could be easily automated in software. The results are being reported for only three cells in Table 5.1. At the time of drafting of this report the integration of this feature within the software simulation environment is still a work in progress and significant speedup in defect simulation time is expected once this change materializes.

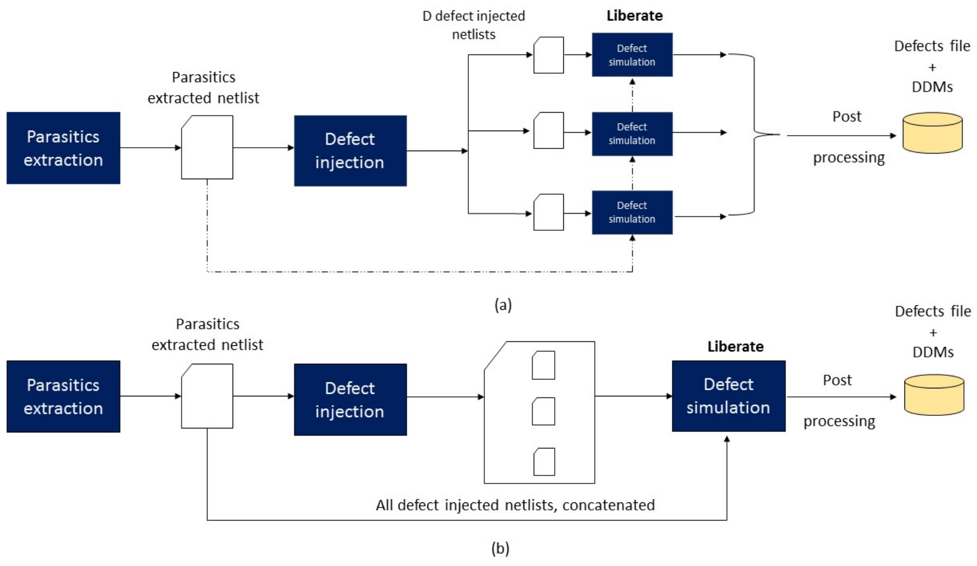


Figure 5.5: Defect simulation (a) using $D+1$ netlists vs. (b) combining all defect injected netlists.

Table 5.1: Speedup - defective netlists combined as sub-circuits within a large netlist.

Cell	**Original approach (D+1) netlists**	**Combining all defect injected netlists in one large file**	**Speedup achieved**
BUFX2	6m 19s	31s	12.2 times
AND2X1	11m 40s	52s	13.5 times
OR2X1	10m 55s	51s	12.8 times

5.7. Inserting a Single Short between Net Pairs

A large coupling parasitic capacitance between two nodes belonging to different nets indicates the nets are very close and run in parallel for longer distances, thereby have a very high chance of being shorted. During parasitic-extraction, parasitic capacitors are extracted between all relevant nodes in the cell. Multiple capacitors could be present between the same net pair, as a result the number of capacitors grows rapidly. This is especially true for large cells such as adders, and multiplexers where the number of extracted parasitic capacitors could be in the order of thousands. If the nodes belonging to a net-pair are shorted, their entire nets are shorted and any additional shorts between other nodes exhibit the same defect behaviour and therefore are equivalent. As illustrated in Figure 5.6, a short between any two nodes of A and B translates to a short between nets A and B, and all such shorts form an equivalent set.

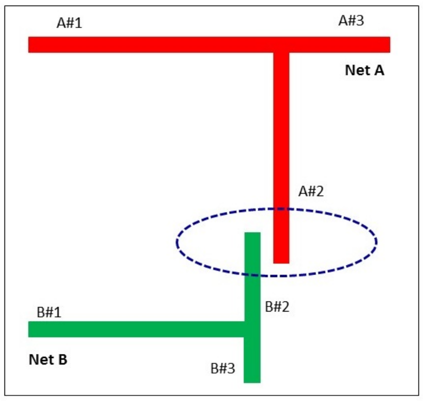


Figure 5.6: Shorts between nodes of a net-pair are equivalent

To minimize the number of short defects and hence the simulation time, it is sufficient to insert a single short between two nets. One way of doing this is to insert a short between two net pairs at the location of the largest parasitic capacitance. Figure 5.7 illustrates the algorithm for inserting a single short between two net pairs. Let *listofCells* be the set of library cells to be considered, such that *cell* ϵ *listofCell*. C_{th} is the user defined threshold value which decides which capacitors are to be considered as possible defect candidates. Per net pair, n_1, n_2 ϵ $Nets_c$ a capacitor between nodes i ϵ n_1 and j ϵ n_2 is selected with the maximum *C(i,j)* such that *C(i,j)* ≥ C_{th}.

```
for all cells ε listofCells do {

    RC parasitics extraction;

        for all $n_1, n_2$ ε $Nets_c$ do {
        netShort($n_1, n_2$ ) = 0;
        }

        for all $n_1, n_2$ ε $Nets_c$ do {
            if netShort($n_1, n_2$) = 0 then {
                shortFound = false;
                for all i ε $n_1$ and j ε $n_2$ do {
                    if C(i,j) ≥ $C_{th}$ then {
                        if ( ⌐(shortFound) ∨ C(i,j)  > C($m_i$,$m_j$) then {
                            shortFound  = true;
                            $m_i$ = i, $m_j$ = j;
                        }
            } } };
            if shortFound then {
                add short defect for node pair ($m_i$,$m_j$) ;
                netShort($n_1$, $n_2$) = 1; netShort ($n_2$, $n_1$) = 1;
            }
```

Figure 5.7: Algorithm for inserting a single short between two net pairs.

5.8. SUMMARY

This chapter discussed various approaches to address the shortcomings identified in the existing library characterization flow by Cadence. Conventional approaches to parasitics extraction are not very effective in generating suitable netlists for cell-aware defect modeling. Therefore, customized settings for generating transitor-level netlists targeting cell-aware defect modelling are proposed. Inclusion of potential defects resulted in inaccurate defect simulations and they had to be eliminated to improve the overall test quality. The threshold values for parasitic resistors and capacitors to be considered for defect insertion is set at 0.0 Ω and 1 x 10^{-19} F. By avoiding the repeated execution of software tasks, a speedup of upto 12 times in defect simulation time is achieved. By inserting a single short defect between two net pairs, the simulation of equivalent defects can be avoided. The experimental results of implementing these modifications to the existing flow are presented in Chapter 6.

6
Experimental Results

This chapter presents the improvements and the experimental results of applying the various modifications presented in Chapter 5.

6.1. Improvements due to elimination of potential defects

To measure the improvement in test quality and simulation time by getting rid of potential defects, the library characterization flow was executed on the GPDK045 library with the following settings:

- *With potential defects - baseline (original) flow by Cadence at the beginning of the project*: the following defect resistance values were used - open = 2 $G\Omega$, short = 0.001 Ω, potential_open = 0.001 Ω, potential_bridge = 2 $G\Omega$.
- *Without potential defects* - after eliminating potential defects and using hard defect values: open = 2 $G\Omega$, short = 0.001 Ω.

The effect of eliminating potential defects on the quality of defect simulation is obtained by comparing the results of characterization *with potential defects* and *without potential defects*. Figure 6.1 illustrates the gain or loss in the number of defect detections for each cell as a result of this modification. A large number of defects changed from being detected to not detected upon eliminating potential defects. On an average each cell lost three detections because of this change. This leads to the conclusion that a large number of defects which were reportedly detected in the original flow, were in fact false detections. Except for the two-input NOR cell (NOR2X2) which had 15 more defects detections, all the other cells either had no new detections or lost detections after this change. For the entire standard-cell library, this translated to a total of 1114 less detections out of a total 37703 defects or approximately 3% difference in defect detection results. Figure 6.2 illustrates this result in terms of the net gain or loss in defect detections. The blue bars represent the fraction of the total defects which are detected with or without potential defects. The grey bars represents those defects which are not detected in either case, and the red bars represent the net loss in defect detections because of eliminating potential defects.

6.2. Improvements due to insertion of super-hard defects

To measure the improvement in test quality and simulation time by inserting super-hard defect values the results of the two runs were compared:

- *Without potential defects* - after eliminating potential defects and using hard defect values: open = 2 $G\Omega$, short = 0.001 Ω.
- *Using super hard defects* - after eliminating potential defects and using hard defect values: open = 1000 $G\Omega$, short = 0.0 Ω.

Figure 6.3 illustrates the gain/loss in the number of defect detections as a result of inserting super hard defects. For the entire standard-cell library an additional 99 defects were detected by changing the defect insertion value from 2 $G\Omega$ to 1000 $G\Omega$ and 0.001Ω to 0.0Ω for opens and shorts respectively.

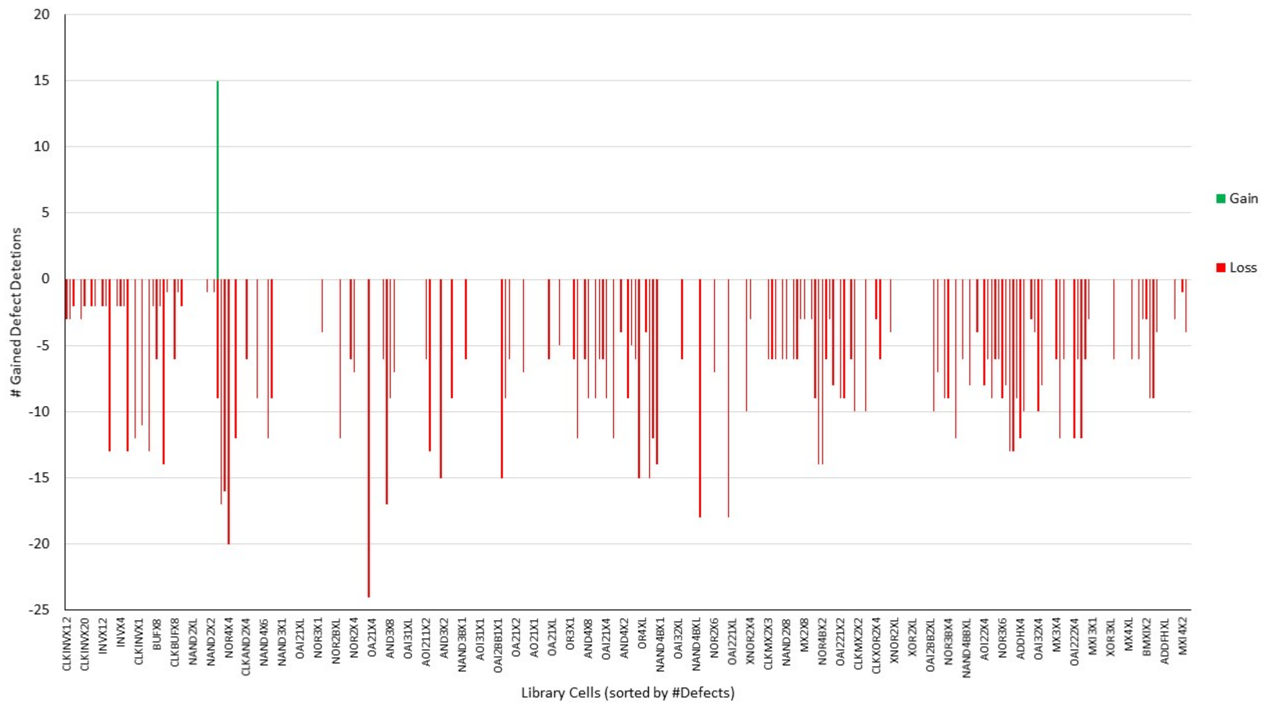


Figure 6.1: Defect detections gained or lost per-cell after eliminating potential defects

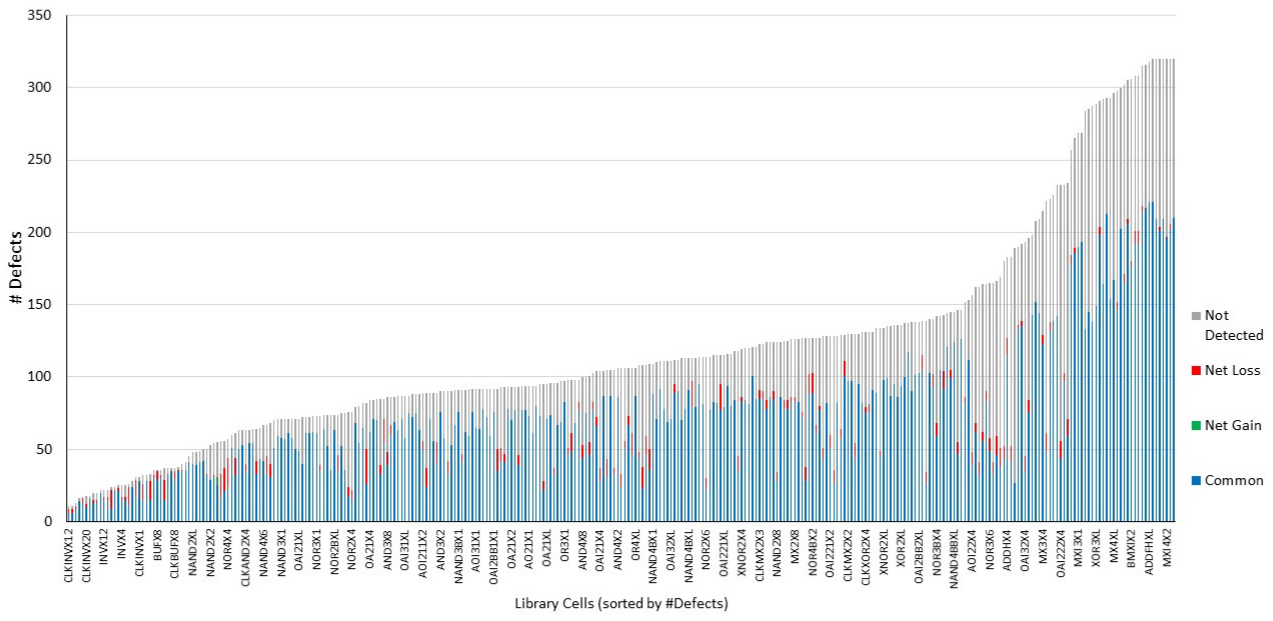


Figure 6.2: Defect detections *with potential defects* and *without potential defects*.

The increase/decrease in the number of test patterns which can detect a defect is illustrated in Figure 6.4. For instance, the library cell numbered 201 (two-input NAND, drive strength of eight - NAND2X8) has four defects which can be detected using 1 additional pattern and two defects which can be detected using 3 additional patterns. But there are five defects for this cell, which have 4 less patterns to detect them. In total, 109 defects had one or more additional test patterns which could detect them, whereas 55 defects had lesser number of

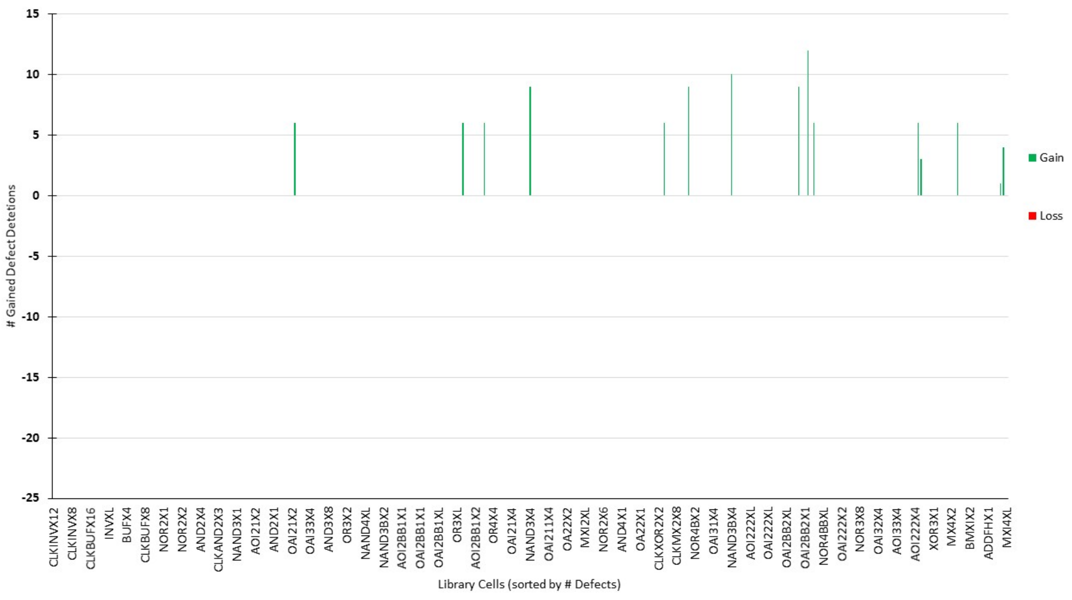


Figure 6.3: Defect detections gained or lost per-cell after inserting super-hard defects

test patterns.

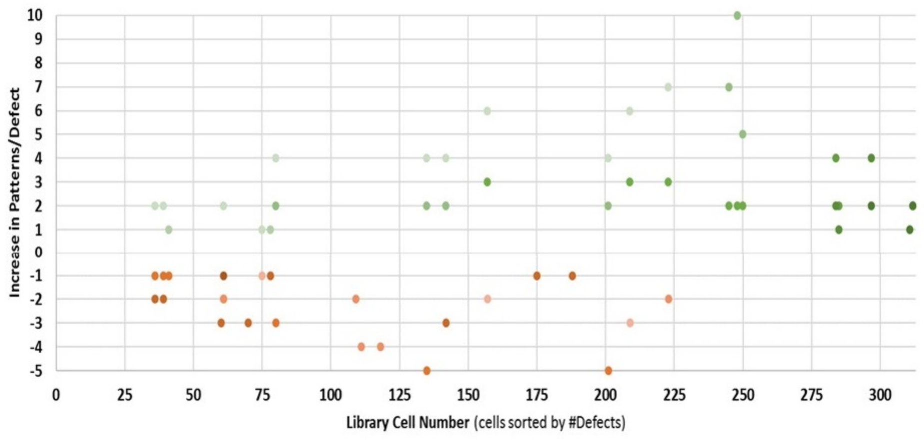


Figure 6.4: Increase or decrease in number of test patterns upon inserting super-hard defects.

6.3. EFFECT ON SIMULATION TIME

To study the impact of eliminating potential defects and inserting very hard defects on the simulation time the simulation times of three characterization runs - (1) with potential defects (2) without potential defects and (3) without potential defects but inserting super hard defects were compared. This data is plotted in Figure 6.5. The simulation time required for characterizing individual cells doesn't vary by large magnitudes after eliminating potential defects or inserting super hard defects. Figure 6.6 depicts the relative difference in the simulation time between the three runs. For instance, on eliminating potential defects the average simulation time per cell decreased by 0.33% (i.e. *without potential defects* vs *with potential defects*). The simulation time reduced by 6.47% when potential defects were eliminated and super hard defects were inserted. In fact for some cells the time required for characterizing defective netlists without potential defects was higher than time required for netlists with potential defects. The initial hypothesis of a large reduction in simulation time, because of simpler netlists did not turn out to be true. In fact, the marginal improvement in simulation time was due to the fact that for D defect candidates, $|D|+1$ defect injected netlists were simulated

(D defective netlists and one fault-free). Every time a defect injected netlist was picked up for simulation, a number of background tasks such as variable initialization, environment setup, parsing the netlist etc. had to be executed. As a result a noticeable improvement in simulation time was not seen in-spite of simplifying the netlist structure prior to simulation. Section 5.6 discussed a possible approach to address this issue.

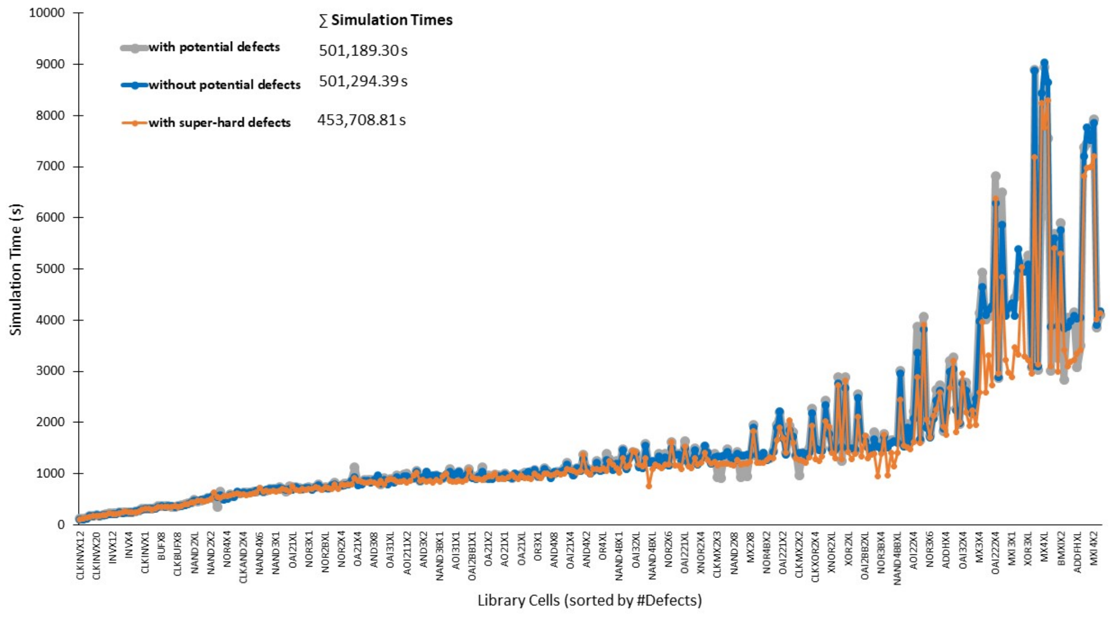


Figure 6.5: Comparison of simulation times.

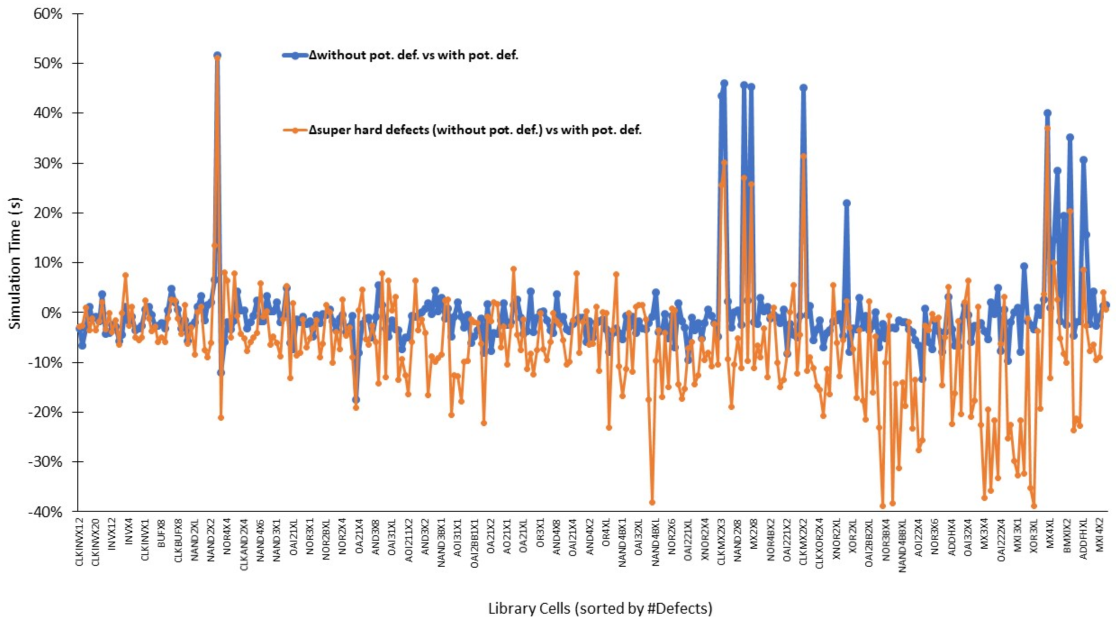


Figure 6.6: Δ simulation times.

6.4. Reduction in number of shorts

To measure the reduction in the number of short defect candidates, because of only inserting a single short between two nets consider the following equations.

$$\text{Let } N \text{ represent the total number of nets in a cell} \tag{6.1}$$

$$n_i \text{ represents the number of internal nodes in net } i \text{ of the cell} \tag{6.2}$$

$$\text{The total number of internal nodes for all the nets, } n_t = \sum_{i=1}^{N} n_i \tag{6.3}$$

$$\text{If a } C \text{ is extracted between every node pair, the number of } C\text{'s } = \frac{n_t(n_t-1)}{2} \tag{6.4}$$

$$\text{Self-capacitors, shorts between nodes belonging to same net } i,\ sc_i\ = \frac{n_i(n_i-1)}{2} \tag{6.5}$$

$$\text{Total number of self-capacitors, all nets} = \sum_{i=1}^{N} \frac{n_i(n_i-1)}{2} \tag{6.6}$$

$$\text{After eliminating self-capacitors, number of shorts} = \frac{n_t(n_t-1)}{2} - \sum_{i=1}^{N} \frac{n_i(n_i-1)}{2} \tag{6.7}$$

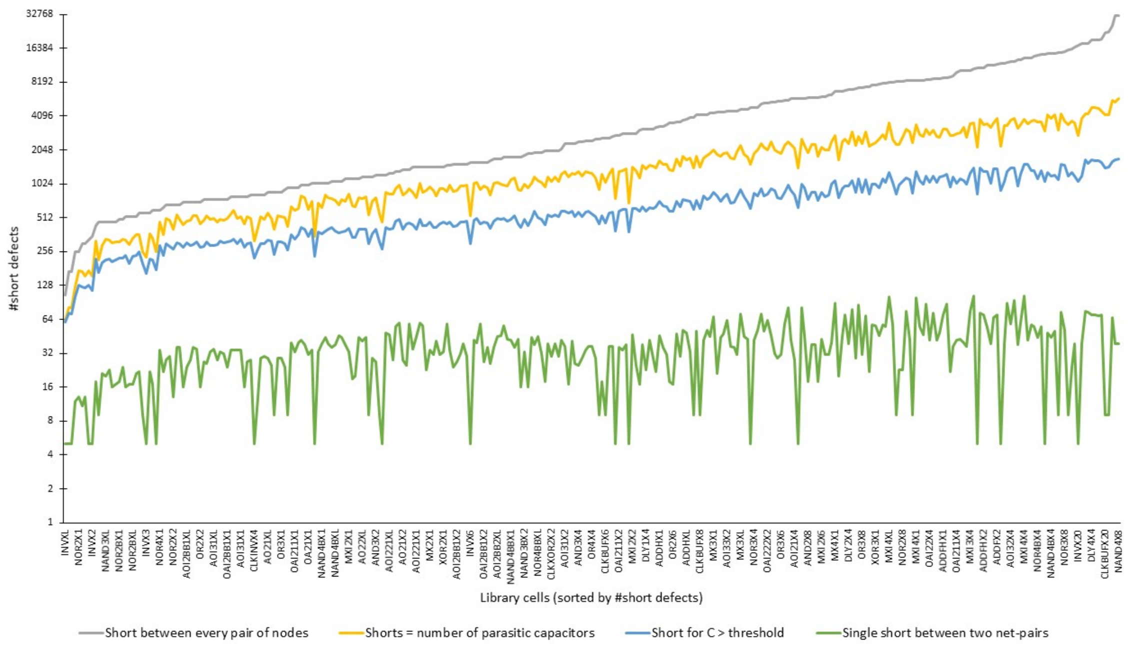


Figure 6.7: Inserting a single short defect per net-pair for all cells.

Figure 6.7 illustrates the comparison between the theoretical analysis and the realistic short defect insertion. The difference between the actual number of extracted parasitic capacitors $C_{extracted}$ and capacitors above threshold $C_{>th}$ grows along the horizontal axis. This is because the cell's complexity increases along the horizontal axis. In simpler cells, due to the small layout of the cell coupling capacitance between two nodes which are far apart is still large enough, therefore the difference between $C_{extracted}$ and $C_{>th}$ is marginal. For complex cells or cells with a higher drive strength the nodes are far apart, and the difference between the actual number of extracted parasitic capacitors and those above the user-defined threshold is large. If a short is injected for every extracted parasitic capacitor in the netlist, the total number of shorts (excluding source-drain shorts) for the entire GPDK045 library are 512201. Filtering out the unlikely shorts and only considering the parasitic capacitors above the $C_{>th}$ of 1 x 10^{-19} F, the number of shorts reduces to 210501, or by 59%. Inserting a single short between two nets for e.g. at the location of the maximum parasitic capacitance, this number drops by a huge margin to 11527. In summary, changing the defect insertion strategy from injecting a short for every parasitic capacitor to a single short between two net pairs, the effective reduction in the number of short defect candidates is 97.7%. This translates to significant improvement in the run-time required for characterizing the library cells.

6.5. ANALYSIS FOR THE GPDK045 LIBRARY

On the basis of the parasitics extraction settings outlined in section 5.2, the number of resistors, capacitors and transistors extracted by the Quantus PEX tool for all the cells in the library are shown in Figure 6.8. A few example cells with the number of transistors, resistors and capacitors extracted are listed in Table 6.1.

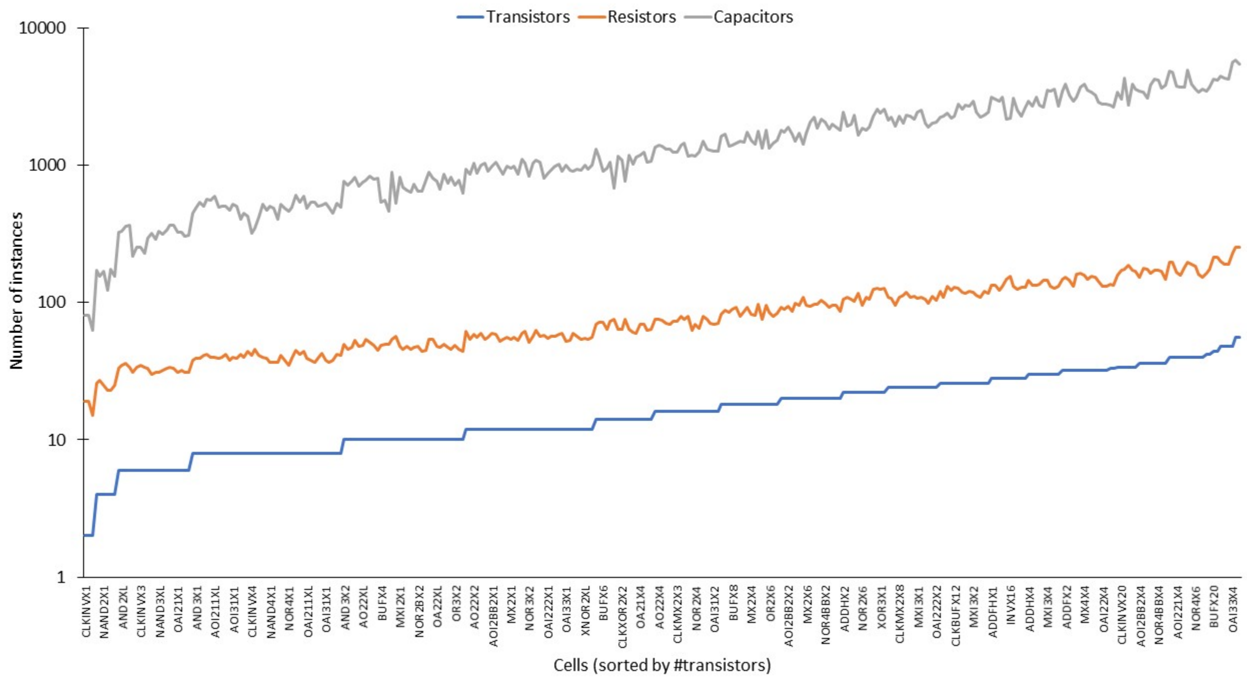


Figure 6.8: Number of transistors, resistors and capacitors for the GPDK045 library.

Table 6.1: Number of transistors, resistors and capacitors extracted for few cells.

Cell	# Transistors	# Resistors	# Capacitors
NOR4X8	56	253	5479
NAND4X8	56	254	5834
ADDFX4	40	198	4804
INVX20	34	187	2734
MX4X2	28	132	3118
BUFX6	14	72	915
CLKINVX6	10	50	466
INVX2	4	27	156

For every transistor three defects - source-open, drain-open and source-drain shorts are inserted. The thresholds for deciding which parasitic resistors or capacitors are to be modeled as defects can be set by the user. For the current flow the threshold for parasitic resistors is set at 0.0 Ω and capacitors is set at 1 x 10^{-19} F. The number of opens, shorts and transistor defects injected for the entire library are illustrated in Figure 6.9. Only a single short is inserted between two pairs at the location of the maximum capacitance. The average number of opens and shorts per cell are 84 and 36 respectively. When we compare the number of extracted parasitic capacitors (possible short defect) with resistors (possible open defect) (Figure 6.8 there is a large difference. Upon inserting a single short between two net pairs compared to injecting an open for every resistor (threshold for parasitic resistance is 0.0 Ω), the number of open defects injected surpasses the number of shorts.

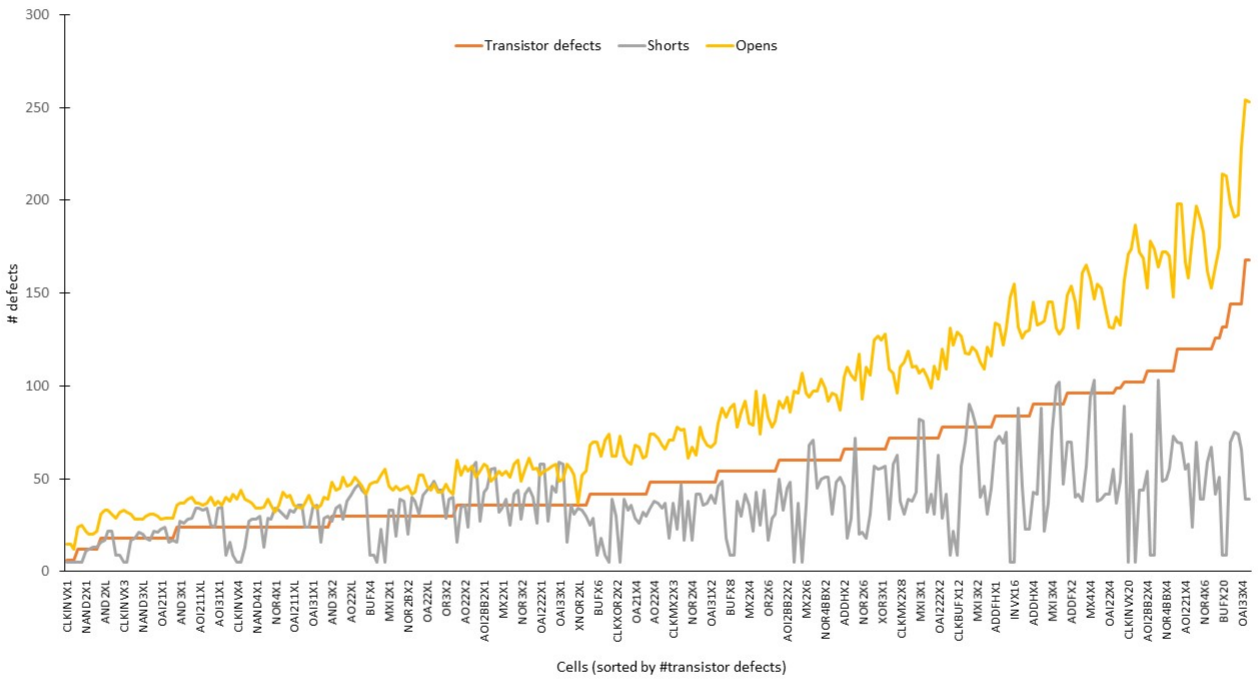


Figure 6.9: Number of transistor, open and short defects.

6.6. SUMMARY

Getting rid of the potential defects led to the identification of 1114 false detections for the entire GPDK045 library. No noticeable improvement in simulation time was observed, as this seemed to be concerned with the design of software flow. This issue was addressed by exploiting the in-built capabilities of the Liberate simulation tool and avoiding repeated execution of boilerplate code. Shorts between two nodes belonging to the same net pair are equivalent. By inserting a single short for every net-pair, the number of short defects to be simulated reduced by 97.7%.

7 Conclusion and Future Scope

The primary objectives of this thesis are to improve the test quality, and to reduce the simulation time for characterizing standard-cell libraries based on the Cadence cell-aware test flow. The thesis started by identifying the following shortcomings in the existing flow.

1. *Parasitics extraction settings:* Cell-aware test is a non-standard application of parasitics extraction tools. This motivated the need for defining customized settings for the parasitics extraction tool for generating transistor level netlists suitable for cell-aware defect modelling.
2. The original parasitic extracted netlists for all the library cells (offered as a part of the RAK database) had several issues: parasitic capacitors were always referenced with respect to GND (V_{ss}), some form of RC reduction was used while extracting the netlists, via resistors were not included, and no diagnostic information was available in the extracted netlists (Chapter 4).
3. *Arbitrary thresholds for defect insertion:* The cut-off values for deciding which parasitic resistors and capacitors are to be considered as defect candidates were arbitrary.
4. *Insertion of potential defects:* Superfluous elements which actually did not exist in the circuit were being inserted as place holder values for possible defect sites. This negatively impacted the quality of the defect simulation results.
5. *Resistance values to be used for modelling defects:* The defect resistance values used for modelling opens and shorts could be made harder.

Apart from these, during the course of executing this thesis several other issues were discovered. These include, drawbacks of using Spectre netlist format, simulation of a separate defect injected netlist per defect, and a software bug related to reversal of transistor source-drain terminals during defect extraction. The thesis work addressed these issues, and the major conclusions derived from this work are the following.

1. *Defining customized settings for the parasitic extraction tool targeting cell-aware test:* Proposal of customized settings for generating parasitics extracted transistor-level netlists which are well-suited for cell-aware defect modelling. (Section 5.2)
2. *Threshold values for open and short defect insertion:* The threshold value for inserting open defects is set at 0.0 Ω, this ensures that maximum number of open defect candidates are covered. Based on a heuristics based approach of correlating the magnitude of the parasitic capacitance between two nets with their locations in the layout, the threshold for inserting a short is set at 1 x 10^{-19} F. (Section 5.3)
3. *Effect of eliminating potential defects:* Potential defects were unwanted elements being inserted into the netlists prior to defect simulation. Getting rid of them uncovered as high as 1114 false detections, out of the total 37703 cell-internal defects possible in the GPDK045 library. This improved the overall quality of the defect simulation results. Elimination of potential defects does not improve the defect simulation time by noticeable margins. (Section 5.4 and Section 5.5)

4. *Impact of inserting super hard defects:* The defect resistance value used for modeling opens was changed form 2 $G\Omega$ to 1000$G\Omega$ and 0.001 Ω to 0.0 Ω for shorts. Inserting super hard defect values resulted in the detection of an additional 99 defects. As a result of inserting super-hard defects, 109 defects had additional test patterns which could detect them. (Section 5.5)

5. *Avoiding repeated execution of background tasks for runtime speedup:* By avoiding repeated execution of boilerplate code such as the initialization of startup environment for Liberate, speedups of upto 12 times in simulation time can be achieved. This requires modifying the software scripts and concatenating all the defect injected netlists together into a single large netlist file. (Section 5.6)

6. *Inserting a short between net-pairs significantly reduces the number of defect candidates:* A short between any two nodes belonging to the same net pair exhibit the same defect behaviour, hence they are equivalent. Carrying out a defect simulation for all such shorts is futile, and a waste of valuable simulation time. Inserting a single short between two net pairs reduces the number of short defect candidates for the entire library by 97.7%. This automatically brings down the simulation time. (Section 5.7)

In conclusion, an overview of the improved flow for library characterization is illustrated in Figure 7.1.

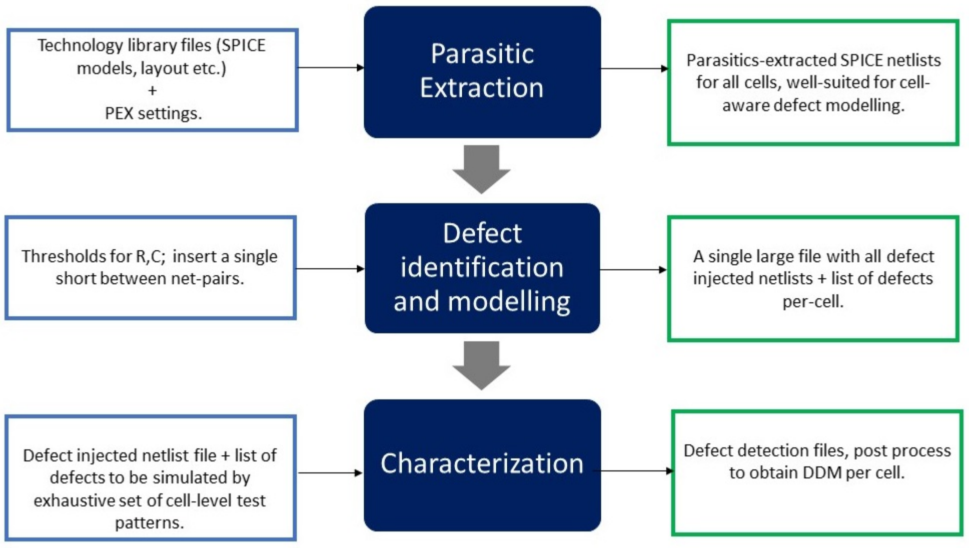

Technology library files (SPICE models, layout etc.) + PEX settings.
Parasitic Extraction
Parasitics-extracted SPICE netlists for all cells, well-suited for cell-aware defect modelling.
Thresholds for R,C; insert a single short between net-pairs.
Defect identification and modelling
A single large file with all defect injected netlists + list of defects per-cell.
Defect injected netlist file + list of defects to be simulated by exhaustive set of cell-level test patterns.
Characterization
Defect detection files, post process to obtain DDM per cell.

Figure 7.1: Improved flow for cell-aware library characterization

The following ideas are suggested as a future scope of work:

- At the time of drafting this thesis, the process of incorporating these modifications into the Cadence cell-aware flow is still a work in progress. The complete flow must be executed on a technology library such as GPDK045.

- To further improve the test quality cell-aware defect modelling must account for timing defects.

- Static simulations yield good quality results and are much faster than analog simulations. However, the analog simulations based approach is more accurate, library characterization flow could be modified to use an analog simulator.

BIBLIOGRAPHY

[1] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI test principles and architectures: design for testability* (Academic Press, 2006).

[2] L.-T. Wang, C. E. Stroud, and N. A. Touba, *System-on-chip test architectures: nanometer design for testability* (Morgan Kaufmann, 2010).

[3] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test* (Morgan Kaufmann, 2009).

[4] L. Lavagno, I. L. Markov, G. Martin, and L. K. Scheffer, *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology: Circuit Design, and Process Technology* (CRC Press, 2016).

[5] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits,* Vol. 17 (Springer Science & Business Media, 2004).

[6] M. R. Prasad, P. Chong, and K. Keutzer, *Why is atpg easy?* in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* (ACM, 1999) pp. 22–28.

[7] J. P. Roth, *Diagnosis of automata failures: A calculus and a method,* IBM journal of Research and Development , 278 (1966).

[8] J. P. Roth, W. G. Bouricius, and P. R. Schneider, *Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits,* IEEE Transactions on Electronic Computers , 567 (1967).

[9] P. Goel, *An implicit enumeration algorithm to generate tests for combinational logic circuits,* IEEE transactions on Computers , 215 (1981).

[10] H. Fujiwara and T. Shimono, *On the acceleration of test generation algorithms,* IEEE Transactions on Computers , 1137 (1983).

[11] F. Hapke, R. Krenz-Baath, A. Glowatz, J. Schlöffel, H. Hashempour, S. Eichenberger, C. Hora, and D. Adolfsson, *Defect-oriented cell-aware atpg and fault simulation for industrial cell libraries and designs,* in *Test Conference, 2009. ITC 2009. International* (IEEE, 2009) pp. 1–10.

[12] I. Pomeranz and S. M. Reddy, *On n-detection test sets and variable n-detection test sets for transition faults,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19**, 372 (2000).

[13] I. Pomeranz and S. Reddy, *Definitions of the numbers of detections of target faults and their effectiveness in guiding test generation for high defect coverage,* in *Proceedings of the conference on Design, automation and test in Europe* (IEEE Press, 2001) pp. 504–508.

[14] J. Geuzebroek, E. J. Marinissen, A. Majhi, A. Glowatz, and F. Hapke, *Embedded multi-detect atpg and its effect on the detection of unmodeled defects,* in *Test Conference, 2007. ITC 2007. IEEE International* (IEEE, 2007) pp. 1–10.

[15] K. Miyase, S. Kajihara, I. Pomeranz, and S. M. Reddy, *Don't-care identification on specific bits of test patterns,* in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on* (IEEE, 2002) pp. 194–199.

[16] K. Y. Cho, S. Mitra, and E. J. McCluskey, *Gate exhaustive testing,* in *Test Conference, 2005. Proceedings. ITC 2005. IEEE International* (IEEE, 2005) pp. 7–pp.

[17] P. Dahlgren and P. Liden, *A fault model for switch-level simulation of gate-to-drain shorts,* in *vts* (IEEE, 1996) p. 414.

[18] F. Hapke, W. Redemund, J. Schloeffel, R. Krenz-Baath, A. Glowatz, M. Wittke, H. Hashempour, and S. Eichenberger, *Defect-oriented cell-internal testing,* in *Test Conference (ITC), 2010 IEEE International* (IEEE, 2010) pp. 1–10.

[19] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast, *Cell-aware test,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **33**, 1396 (2014).

[20] F. Zhang, M. Thornton, and J. Dworak, *When optimized n-detect test sets are biased: An investigation of cell-aware-type faults and n-detect stuck-at atpg,* in *Test Workshop (NATW), 2014 IEEE 23rd North Atlantic* (IEEE, 2014) pp. 32–39.

[21] F. Hapke and J. Schloeffel, *Introduction to the defect-oriented cell-aware test methodology for significant reduction of dppm rates,* in *Test Symposium (ETS), 2012 17th IEEE European* (IEEE, 2012) pp. 1–6.

[22] F. Hapke, J. Schloeffel, H. Hashempour, and S. Eichenberger, *Gate-exhaustive and cell-aware pattern sets for industrial designs,* in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on* (IEEE, 2011) pp. 1–4.

[23] F. Hapke, M. Reese, J. Rivers, A. Over, V. Ravikumar, W. Redemund, A. Glowatz, J. Schloeffel, and J. Rajski, *Cell-aware production test results from a 32-nm notebook processor,* in *Test Conference (ITC), 2012 IEEE International* (IEEE, 2012) pp. 1–9.

[24] F. Hapke, R. Arnold, M. Beck, M. Baby, S. Straehle, J. Goncalves, A. Panait, R. Behr, G. Maugard, A. Prashanthi, *et al.*, *Cell-aware experiences in a high-quality automotive test suite,* in *Test Symposium (ETS), 2014 19th IEEE European* (IEEE, 2014) pp. 1–6.

[25] A. Touati, A. Bosio, L. Dilillo, P. Girard, A. Virazel, P. Bernardi, and M. S. Reorda, *Scan-chain intra-cell defects grading,* in *Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2015 10th International Conference on* (IEEE, 2015) pp. 1–6.

[26] A. Touati, A. Bosio, P. Girard, A. Virazel, M. S. Reorda, E. Auvray, *et al.*, *Scan-chain intra-cell aware testing,* IEEE Transactions on Emerging Topics in Computing (2016).

[27] S. Potluri, A. Mathew, R. Nerukonda, I. Hartanto, and S. Toutounchi, *Cell-aware atpg to improve defect coverage for fpga ips and next generation zynq® mpsocs,* in *Asian Test Symposium (ATS), 2017 IEEE 26th* (IEEE, 2017) pp. 157–162.

[28] H.-W. Liu, B.-Y. Lin, and C.-W. Wu, *Layout-oriented defect set reduction for fast circuit simulation in cell-aware test,* in *2016 IEEE 25th Asian Test Symposium (ATS)* (IEEE, 2016) pp. 156–160.

[29] *Encounter Test Cell-Aware Lab Instructions, 2014-15*, Cadence (2014).

A
CODE LISTINGS

```
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#         Listing 1 - Top level Script for Executing Cell-Aware Flow                        #
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

#!/usr/bin/python

import sys
import os
import time

# Reads the list of cells from listofCells.txt (in USER.INPUTS)
# writes a cell to target.cellList
# executes the netlist+characterization flow by invoking runmodus.library.cellaware script
# creates a seperate log file for each cell with information about real, user and system time.
# lisofCells.txt in /USER.INPUTS which now contains the list of cells
# along with index.txt which tracks the number of cells being characterized
# do not write the list of cells in target.cellList if using this script


# read previous index (a way to keep track of number of cells being processed)
cwd = os.getcwd()
indexFileName = "index.txt"
indexFile = os.path.join(cwd,'USER.INPUTS', indexFileName)
indexContents = open(indexFile, 'r').readline().strip()
index = int(indexContents)

# read the list of cells specifed in listofCells.txt
cwd = os.getcwd()
# print "Current working directory: %s" % (cwd)
cellListFileName = "listofCells.txt"
cellListFile = os.path.join(cwd,'USER.INPUTS', cellListFileName)
CellListFileContents = open(cellListFile, 'r').readlines()
print "listofCells.txt contains %s \n" % (CellListFileContents)

for Cell in CellListFileContents:
  cwd = os.getcwd()
  # print "Current working directory: %s" % (cwd)
  cell = Cell.strip()
  print "Current Cell under consideration: %s \n" % (cell)

  # write cell to target.cellList which will be input to modus script
  targetCellList = "target.cellList"
  targetCellListFile = os.path.join(cwd,'USER.INPUTS',targetCellList)
  targetf = open(targetCellListFile,'w')
  targetf.write('%s' % cell.strip())
  targetf.close()
  print "%s cell written to target.cellList file \n" % (cell)
  time.sleep(1)

  # to check contents in target.cellList file
  targetfContents = open(targetCellListFile, 'r').readline()
  cell = targetfContents.strip()
  print "target.cellList contains %s \n" % (cell)
  targetf.close()

  # start executing for each cell
  # to prevent overwriting of the logs results per cell are stored in a seperate log file
  logFile = "%s_%s" % (index,cell)
  # print logFile
  # cmd = "(time modus -file ./SCRIPTS/netlist_only.tcl) &> %s" % (logFile)
  cmd = "(time modus -file ./SCRIPTS/runmodus.library.cellaware.tcl) &> %s" % (logFile)
  print "Executing %s \n" % (cmd)
  os.system(cmd)

  # move the log file to /RESULTS/LOGS directory
  cwd = os.getcwd()
  # print cwd
  resultsDir = os.path.join(cwd,'RESULTS', 'LOGS')
  # print resultsDir
  cmd = "mv %s " \
    "%s" % (logFile, str(resultsDir))
  print "Executing %s \n" % (cmd)
  os.system(cmd)

  # copy the DDM to /RESULTS/DDM directory
  cwd = os.getcwd()
  # print cwd
  resultsDir = os.path.join(cwd,'RESULTS','DDM')
  # print resultsDir
  ddmFile = cell + '.txt'
  ddmDir = os.path.join(cwd,'cell_aware_files','2_characterized_cell_faults',cell,'detections', ddmFile)
  # print ddmDir
  cmd = "cp " \
    "%s " \
    "%s/" % (str(ddmDir), str(resultsDir))
  print "Executing %s \n" % (cmd)
  os.system(cmd)

  # copy the OR GROUPS to /RESULTS/OR_GROUP directory
  cwd = os.getcwd()
  # print cwd
  resultsDir = os.path.join(cwd,'RESULTS','OR_GROUPS')
  # print resultsDir
  orFile = cell + '.gz'
  orDir = os.path.join(cwd,'cell_aware_files','3_prepared_cell_faults','cell_aware_fault_rules', orFile)
  # print orDir
  cmd = "cp " \
    "%s " \
    "%s/" % (str(orDir), str(resultsDir))
  print "Executing %s \n" % (cmd)
  os.system(cmd)
  index+=1

# write index to a file (a way to keep track of number of cells being processed)
indexFileName = "index.txt"
cwd = os.getcwd()
indexFile = os.path.join(cwd,'USER.INPUTS', indexFileName)
indexf = open(indexFile, 'w')
indexf.write('%s' % index)
indexf.close()
```

```
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#          Listing 2 - Script for Automating Quantus Parasitics Extraction                         #
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

#!/usr/bin/python

import sys
import os
import time

```

```

################ CAUTION ! ####################################
# Once you run a complete QRC flow a ccl file is created in svdb directory
# Modify the input, output and svdb directory paths in generated ccl file
# copy and rename it to optimum_qrc_settings.ccl' and place it in top level directory
# in batch mode change all paths in optimum_qrc_settings.ccl to ./autoPVS
# top level directory = directory from were virtuoso is invoked
# Create "SPICE_Extracted"

# Assumes that PVS-LVS is already run and required input data for QRC is available
# in directory e.g. - ./autoPVS (if PVS-LVS is run using batch mode) or ./LVS-PVS
# At every iteration it replaces the name of the cell in optimized_qrc_settings.ccl and
# executes the QRC-PEX flow
# This step is repeated for all the cells whose DRC-LVS and QRC input data is available,
# and the have been specified in the paths
# the trick is to use 'sed - stream editor command in linux'
# place a file 'listofCells' with the name of all the cells on which QRC extraction
# is to be performed in the current working directory

# Initial ccl file is generated using AND2X1 cell, if initial ccl file is generated
# with a different cell name then change accordingly
initialCellName = 'AND2X1'

cwd = os.getcwd()
print "Current working directory %s" % (cwd)

# Read the list of cells from a file on which to perform a QRC extraction
cellListFileName = 'listofCells'
cellListFile = os.path.join(cwd,cellListFileName)
listofCells = open(cellListFile,'r').readlines()
# print "List of Cells contains:"
# print listofCells

path = "./autoPVS"
# path = "./LVS_PVS"

for cell in listofCells:
  # replace the cell name in generated ccl file with new cell name
  cellName = cell.rstrip()
  print "Generating ccl file for %s\n" % (cellName)
  cmd = "sed -i 's/%s/%s/g' optimum_qrc_settings.ccl" %(initialCellName,cellName)
  print "%s \n" % (cmd)
  os.system(cmd)

  #run qrc extraction using the ccl file which has a new cell name
  print "QRC Extraction being performed for %s cell" % (cellName)
  cmd = "qrc -cmd optimum_qrc_settings.ccl"
  print "%s \n" % (cmd)
  os.system(cmd)
  initialCellName = cellName
```

B
CADENCE CELL AWARE TEST - RAK DIRECTORY STRUCTURE

Table B.1: The cell-aware flow RAK directory structure.

Directory	Contents	Description
SCRIPTS (input)	liberatesetup.tcl	initialize and setup the environment for Liberate startup
	rcreportLibCells.tcl	report the library cells supported for cell-aware testing
	runmodus.library.cellaware.tcl	top-level script to start cell-aware library characterization
	runrc.report.comb.cells.sh	report the combinational cells supported by cell-aware testing
	runspectre.spp.convert.sh	script to convert netlists from SPICE to Spectre format
USER. INPUTS (input)	GPDK045_45nm_Library + Parasitics extracted transistor-level netlists in Spectre format	Process Design Kit containing the standard-cell layouts, SPICE models, device parameters, rule-files, technology data, parasitic extracted Spectre netlists (fault-free) etc.
	usermodels.scs	file to point to the devie models
	target.cellList	list of cells to be characterized
	spectre_model_includes.spi	path to include Spectre models
	fast_vdd1v2_basicCells.lib	device parameters and definitions required during Liberate characterization
cell_aware_files (output)	1_extracted_cell_faults	defect injected netlists and list of defects file
	2_characterized_cell_faults	results of characterizing the defect injected netlists
	3_prepared_cell_faults	fault rules for cell to be used during ATPG

C
CADENCE LIBRARY CHARACTERIZATION - SOFTWARE EXECUTION

The Modus software environment provides three ways of running the library characterization flow [29]:

1. *Command line or batch mode:* uses a script to initialize the execution environment, and then invokes the Modus shell to execute the flow.
2. *Interactive mode:* executes the entire flow in Modus shell environment.
3. *Graphical User Interface (GUI) Mode*: run the flow by using point and click menu options.

The current project is carried out using the *command line or batch mode*. To start the library characterization flow, from the top-level directory execute:

```
modus -f ./SCRIPTS/runmodus.library.cellaware.tcl
```

The various sub-tasks executed by *runmodus.libary.cellaware.tcl* script are as follows:

1. *Define user-variables:* user-specific variables *(emphasized in italics below)* are initialized to point to various directory and file locations. These include:
   - variables which hold the paths to the current working directory and output directories - *workdir, OUTDIR*
   - path to point to the file containing the list of cells to be characterized (target.cellList file) - *TARGET_CELL_LIST*. In the current flow only combinational cells are supported.
   - path to the transistor model files (in Spectre or SPICE format) - *DEVICE_MODELS*.
   - define - *(1) ecif_outdir* (path to the directory which stores the defect injected netlists), (2) *ccif_outdir* (directory which stores the output of characterizing the defective netlists) (3) *pcif_outdir* (directory which stores the fault rules file to be used during ATPG).
2. *Verify required tools are available in the system path:* The successful execution of the library characterization process depends on several tools. It is very important to verify that all the required tools can be invoked and they operate without any glitches. This prevents possibilities of software errors later in the flow. Some of the common issues during this step might include - incorrect definitions of executables in the system path and conflicts due to different Open Access (OA) database versions (format for standard-cell layouts) supported by the tools. Such issues must be zoomed-in and resolved before the rest of the flow can be executed.
3. *Clean up leftover files from previous runs:* The cell-aware flow generates several files and temporary directories which might occupy significant memory. This step deletes any leftover files and directories from a previous run.

4. *Setup for extracting cell-internal defects:* Define the path to directory containing the parasitics extracted Spectre netlists - *ECIF_SPECTRE_DIR*. The values of the different constants which will be used during defect extraction process are defined or initialized:

   - *Threshold values for resistance(R) and capacitance(C): ECIF_RES_THRESH and ECIF_CAP_THRESH* are initialized to the the user defined threshold values for modelling a parasitic resistor as an open and a parasitic capacitor as a short. Parasitic elements with resistance or capacitance values lesser than these thresholds are not considered as possible locations for a defect, but they remain a part of the defect injected netlist. The default values for *ECIF_RES_THRESH* and *ECIF_CAP_THRESH* are 100 Ω and $3e-17$ Ω respectively. Only these R's and C's will be propagated as defect candidates in the further flow. However, for every transistor three defect candidates are always considered - source-open, drain-open, and source-drain short.
   - *Resistance values for modelling open and short defects: ECIF_ROPEN and ECIF_RBRIDGE* are the values to be used for modelling an open and short (bridge) defect respectively. An open defect candidate (parasitic resistor above *ECIF_RES_THRESH*) is modelled using a very high resistance value. A short (bridge) defect (a parasitic capacitor above *ECIF_CAP_THRESH*) is modelled with a very low resistance in parallel across a parasitic capacitor. The default values for *ECIF_R_OPEN* and *ECIF_R_BRIDGE* are 2 $G\Omega$ and 0.001 Ω respectively.
   - *Potential open and potential bridge resistances: ECIF_POTENTIAL_OPEN and ECIF_POTENTIAL_ BRIDGE* are the resistance values used to negate the defect behaviour at the defect location. For example, to negate the effect of an open the transistor terminals it is replaced with a very low resistance value *ECIF_POTENTIAL_OPEN*. Similarly, a short (bridge) defect is replaced with *ECIF_POTENTIAL _BRIDGE* - a high resistance to eliminate it's effect. The default values for *ECIF_ POTENTIAL_OPEN and ECIF_POTENTIAL _BRIDGE* are 0.001 Ω and 2G Ω respectively.

5. *Setup for Liberate characterization:* Initialize *CCIF_LIBERTY* with the path to the file from which cell I/O and pin names can be determined (fast_vddlv2_basicCells.lib). This information is used by Liberate while reading the cell-netlist for fault simulation. Set *CCIF_TEMPERATURE and CCIF_VOLTAGE* for the temperature and cell-operating voltage, default values are 25 °C and 1.2 V respectively. Define *CA_LIBERATE_SETUP* to point to the file *liberate.setup.tcl*, and execute it to initialize the environment variables required for Liberate startup.

6. *Defect injection and generation of defective netlists:* Using the parasitic extracted Spectre netlists, transistor models, and user specified threshold values, this step creates a defects list file and a defect injected netlist per candidate defect. For every cell the outputs are written to a separate directory under *./cell_aware_files/cellName/01_extracted_cell _faults/cellName.scs*. For every cell specified in *target.cellList* file pointed by *TARGET_CELL_LIST*:

   (a) read the parasitic extracted cell-netlist.

   (b) build a list of Spectre (or SPICE) primitives to identify the transistor models.

   (c) identify the parasitic resistors which are above the user defined thresholds *ECIF_RES_ THRESH*. For every such occurrence replace the resistor with *ECIF_ROPEN* value and add it to the list of open defects. An example defects list file is illustrated in Figure 4.4. The fault effect of all other candidate defects other than the one being considered is negated by replacing them with *ECIF_ POTENTIAL_OPEN and ECIF_POTENTIAL_BRIDGE* as discussed earlier and is illustrated in Figure 4.2.

   (d) identify the parasitic capacitors which are above the user defined thresholds *ECIF_CAP_THRESH*. For every such occurrence replace the capacitor with *ECIF_RBRIDGE* value and add it to the list of short defects. The effect of all other defects other than the one being considered is negated by replacing them with *ECIF_POTENTIAL_OPEN and ECIF_POTENTIAL_BRIDGE* as illustrated in Figure 4.3.

   (e) for every transistor instance generate three faulty netlists to cover source-open, drain-open, and source-drain short defects. To model an open insert *ECIF_ROPEN* on the transistor terminals. For shorts insert a *ECIF_RBRIDGE* resistor between source and drain terminals. Record the list of defects in the defects list file.

(f) generate a separate defect injected netlist for each identified candidate defect. At the end of this step, we are left with $|D|+1$ netlists and (D defect injected netlists, and 1 defect-free) along with a defects list file.

7. *Characterize the target defect set:* Using the defect inserted netlists and the defects list file generated in the previous step, Liberate characterizes these defects by performing a static-simulation. For an n input cell, each defective netlist is simulated with 2^n patterns. A defect is said to be detected if the output response and expected response for a test pattern are different. Otherwise a test pattern cannot detect the defect. Liberate writes this result to a defect detection file per cell as illustrated in Figure 4.5. The defect detection file records only those defects which can be detected. A defect can be detected by more than one cell-level test patterns. Similarly, a cell-level test pattern can detect more than one defects. For every defect which can be detected by at least one test pattern the defect detection file contains the following information:

   - defect number and the description of the defect such as defect location and the nature of the defect (open/short).
   - input test-patterns which can detect the defect.
   - response of the defect-free and defective netlist.

   An example defect detection file for two-input AND cell is illustrated in Figure 4.5. In this example defect number 9 is a short between the nodes \1\:A and V_{SS}. This defect is detected by only one test pattern 11 applied to the inputs. The response of the defect-free case is 1 and the and the faulty response is 0. Because the responses of the defect-free and defective cases are not equal the defect can be detected by this test pattern. It can be seen that defect number 3 is detected by three test patterns, whereas defect 1 and defect 2 are not detected by any test pattern and therefore are not written to the defect detection file.

8. *Prepare cell-internal fault rules for ATPG:* In this step Modus creates a faults-rule file describing the testable faults for each cell. The faults rule file is defined using the Modus Pattern Fault Modelling language. This is Cadence proprietary terminology, and is used by the Modus ATPG engine during circuit level test-pattern generation (CA-ATPG stage).