Constrained-Random Stimuli Generation for Post-Silicon Validation

CONSTRAINED-RANDOM STIMULI GENERATION FOR POST-SILICON VALIDATION

BY

XIAOBING SHI, B.Eng., M.A.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2016)	McMaster University
(Electrical & Computer Engineering)	Hamilton, Ontario, Canada

TITLE:	Constrained-Random Stimuli Generation for Post-Silicon Validation
AUTHOR:	Xiaobing Shi B.A.Sc., (Computer Science and Technology) Huazhong University of Science and Technology, Wuhan, China M.A.Sc., (Computer Science and Engineering) University of Chinese Academy of Sciences, Beijing, China
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xx, 150

To my beloved parents

献给我的父亲母亲

Abstract

Due to the growing complexity of integrated circuits, significant efforts are undertaken to ensure the design and implementation meet the specification and quality requirements both at the pre-silicon verification stage (before tape-out), as well as at the post-silicon validation stage (on the silicon prototypes). In particular, the constrained-random methods, which subject the design to a large volume of random, yet functionally-compliant stimuli, are widely employed during the pre-silicon stage. Hardware description languages, such as SystemVerilog, have standardized and well-defined features to formalize the constraints including format, sequence control and distribution. Nonetheless, it is not obvious how such features can be efficiently leveraged at the post-silicon stage.

In this dissertation, a systematic methodology is proposed to support constrained-random generation and application during post-silicon validation. This includes both software algorithms and on-chip hardware structures. The proposed software translates functional constraints from SystemVerilog into a cube-based representation. A method to design in-field programmable signal generators, which are placed on-chip, can directly expand compacted cubes to extensive random, yet functionally compliant, sequences for post-silicon validation. This approach is extended to also support sequential constraints, as well as the control of the stimuli distribution.

Acknowledgements

I would like to appreciate the help of my supervisor, Dr. Nicola Nicolici, who not only mentors me on the way of researching, but also teaches me the art of leading a better life in professional circles. I am also appreciative to Dr. Mark Lawford and Dr. Shahram Shirani who serve in my supervisory committee. Their broad view and advice on my research topic helped me to better position the work in the dissertation.

I would like to express my thanks to the colleagues, including Adam Kinsman, Philip Kinsman, Zahra Lak, Henry Ko, Linyan Liu, Pouya Taatizadeh, Jason Tong and Amin Vali. Without the communication and help of them, I could not have developed so many insights in the field of post-silicon validation.

I am also thankful to the administrative and technical staff from the Electrical and Computer Engineering Department at McMaster for their work and assistance during my studies.

List of Abbreviations

ASIC	Application Specific Integrated Circuit
ATPG	Automated Test Pattern Generation
BIST	Built-In Self-Test
BDD	Binary Decision Diagram
CAD	Computer-Aided Design
CBC	Compact Binary Cube
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CMOS	Complementary Metal-Oxide Semiconductor
CRC	Cyclic Redundancy Check
CRSG	Constrained-Random Stimuli Generator
DFT	Design For Test
DNF	Disjunctive Normal Form
DRC	Design Rules Checking

DUT	Design Under Test
DUV	Design Under Validation
EDA	Electronic Design Automation
ERC	Electrical Rule Checking
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
HDL	Hardware Description Language
IC	Integrated Circuit
I/O	Input/Output
ILP	Integer-Linear Programming
IP	Intellectual Property
JTAG	Joint Test Action Group
K-map	Karnaugh Map
LFSR	Linear Feedback Shift Register
LVS	Layout Versus Schematic
PC	Personal Computer
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express

PRNG	Pseudo-Random Number Generator
RAM	Random-Access Memory
RTL	Register-Transfer Level
RTP	Real-time Transport Protocol
SAT	Boolean Satisfiability Problem
SoC	System on a Chip
TAP	Test Access Port
TLP	Transaction Layer Packet
VLSI	Very Large Scale Integration

Contents

List of Tables

List of Figures

List of Codes

Chapter 1

Introduction

Integrated circuits (ICs) have been gradually impacting every aspect of human life. We lead better lives because of electronic gadgets, which are powered by ICs, ranging from infotainment systems to biomedical devices. Since today's lifestyle is conditioned by the assumption that these electronic devices which surround us are trustworthy, an important question is whether these devices have been developed, implemented and manufactured as expected, and to what extent do they complete their tasks correctly? It can be argued that making mistakes is rooted in the human nature, hence what measures are taken during the design, implementation and manufacturing of these electronic devices to address any potential failures? Despite their easy-to-use appearance, most of the electronic systems have thousands to even millions of basic building blocks and validating that all of them are implemented correctly is a challenge of practical relevance that must be addressed by systematic approaches.

This chapter provides a high-level overview the implementation cycle of IC designs, especially the processes which help ensure that the stated and implied goals are met. It also positions the contributions from this thesis within the IC implementation cycle.

1.1 IC design and verification tasks

Integrated circuits consist of a set of electronic devices, e.g., transistors, that are integrated onto wafers manufactured from semiconductor material, such as silicon. The first integrated circuits were manufactured in the middle of the 20th century and they were comprised of a few transistors (Kilby, 2001), and from then onwards their complexity has rapidly evolved. The scale of ICs has grown from a few transistors, i.e., small-scale integration and medium-scale integration in the early stages, to large-scale integration and very large-scale integration (VLSI), generally at tens of thousands transistors and above (Smith, 1997). While today's circuits are still referred to as VLSI circuits, it is worth mentioning that state-of-the-art advanced designs, e.g., multi-core processors and system-on-a-chip devices, can have in excess of one billion transistors.

Generally, an IC product goes through several phases by evolving from its concept form to the design form to the fabricated form. First, the specification of the product is defined according to market research and technical goals, for example, what performance the product is expected to achieve, to which standard it has to conform to, or what non-functional requirements (e.g, power consumption) must be met. Thereafter, the product is implemented (most commonly) in a register-transfer level (RTL) model using hardware description languages (HDLs) according to the specification, and subsequently it is synthesized to a netlist and then into the layout of the IC. Up to this point, the product still remains in a virtual form which has not been materialized into a physical device. Therefore it is known as the *pre-silicon* phase. Subsequently the IC is fabricated onto the silicon wafer and any tasks performed after this step are said to operate during the *post-silicon* phase.

In order to guarantee the consistency between the different forms of the product, a series of tasks are employed at both the pre-silicon and post-silicon phases. In general, the methods for verifying the consistency between the design implementation and its specification that are carried out during the pre-silicon phase are categorized as *pre-silicon verification*. *Manufacturing test* ensures that each fabricated device matches its implementation. *Post-silicon validation* is performed on silicon prototypes that have passed manufacturing test, however it is necessary to establish with higher confidence than during pre-silicon verification whether the specification is met. It is a critical step that needs to detect design errors that must be fixed in the subsequent respins before committing to high-volume manufacturing. As shown in Figure 1.1, although these three tasks (pre-silicon verification, manufacturing test and post-silicon validation) have their unique objectives and methods, they complement and reinforce each other to guarantee the quality of the fabricated ICs. The following three sections provide a brief overview of these three tasks and outline some key methods that are used for each of them.

1.2 Pre-silicon verification

Pre-silicon verification is employed to ensure the consistency between the design and its specification. Its goal is not only confined to finding errors in the design but it also uses a systematic way to increase the confidence that the design is able to accomplish the expected functionality (Spear and Tumbush, 2012). The design flow can be seen as the forward process from the abstract concept and specification towards the concrete design structure ready for manufacturing. The verification can be seen as the checking process between different steps in the design flow.

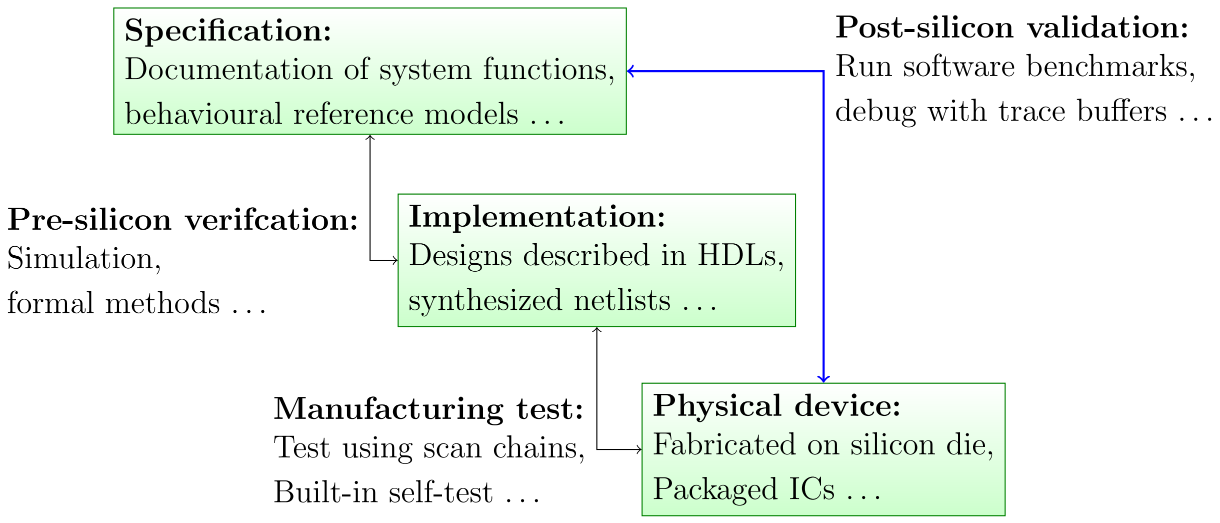

Specification:
Documentation of system functions,
behavioural reference models . . .
Post-silicon validation:
Run software benchmarks,
debug with trace buffers . . .
Pre-silicon verifcation:
Simulation,
formal methods . . .
Implementation:
Designs described in HDLs,
synthesized netlists . . .
Manufacturing test:
Test using scan chains,
Built-in self-test . . .
Physical device:
Fabricated on silicon die,
Packaged ICs . . .

Figure 1.1: Pre-silicon verification, manufacturing test and post-silicon validation: three steps that ensure the consistency between specification, implementation and fabricated devices.

Simulation-based verification and formal verification are used to detect and fix functional errors, followed by physical verification which checks the electrical characteristics of the layout, including timing and power, before committing to the fabrication of silicon prototypes.

1.2.1 Simulation-based verification

A software simulator is a tool that imitates the behaviour of the IC. Given a model of the design (e.g., RTL description or gate-level netlist), the simulator can control the inputs to inject a series of stimuli, and then it can observe both the output response and the state transitions. There are two broad approaches to generating the stimuli: direct and random. Concerning the observation, the responses can be compared against a golden model (most commonly obtained at a higher level of design abstraction), and design properties or assertions can be be monitored.

In direct verification, the expected responses are known by computing them using the same set of input stimuli on a different model of the design, commonly referred to as the reference model. Most commonly this reference model is a description at a higher-level of design abstraction. For example, for an image decoder, a software model (e.g., C/C++ or Matlab) can be used to compute the values of each pixel. The hardware model (e.g., RTL description or gate-level netlist) uses the exact same input stimuli as the software model and therefore the output responses are expected to match. Alternatively, the reference model can be a previous version of the design that has been already prototyped or fabricated (e.g., a microprocessor that is compatible in terms of machine code) and the output responses of the reference model can be computed rapidly on the physical device.

For random verification, the auxiliary software within the simulator, known as the pseudo-random number generator (PRNG), undertakes the work of creating stimuli for the design. The pure random method generates stimuli autonomously, which are used to check simple properties, such as the connections of data paths. Most commonly, the specification of a design has defined a series of formats and rules for the inputs. For an image decoder, for example, the header of the file must have a set of fields and the values within this fields are constrained and can be related to each other. Another example are communication protocols where the packets follow a pre-defined format. The values within some fields can be randomized, however the value in one field might constrain the values in other fields. Hence the stimuli for the design should not be completely random, but random to such an extent as to be constrained within a *valid* or *functionally-compliant* subspace of input values. The generation of such stimuli is referred to as *constrained-random stimuli generation*. In this methodology it is not mandatory to compute golden responses. Rather, because the constraints on the input space guarantee valid stimuli, the objective is to ensure that design properties are not violated.

It is worth mentioning that random verification is not a replacement for direct verification. Rather, it complements it. Both direct and random methods are useful to detect and fix design errors (or bugs), hence they can help answer the question "does-it-work?". If no design errors have been detected after many directed use cases have been employed, the random method is particularly useful to answer the question "are-we-done?". This is because before tape-out what can be measured is limited by the simulation time and accuracy, and designs are released for manufacturing when the confidence level is deemed sufficient. This confidence level is quantified using simulation metrics, such as code or assertion coverage.

1.2.2 Formal verification

Regardless of the metrics that have been employed during simulation-based verification, since the confidence level is quantified by coverage metrics, there is no guarantee that the design is functionally *correct* for the entire valid input space (unless the entire space can be enumerated, which is practically infeasible for VLSI designs). Formal methods, e.g., model checking, can prove the equivalence between the target representation (usually the low-level system model or implementation) and the reference representation (high-level design or specification). The tools for model checking verify the consistency between the model formalized from the low-level design and the properties extracted from the specification. If these two representations (or models) are not equivalent, model checking can produce a counterexample that points out to what lead to the inconsistency between the two models (Clarke *et al.*, 1999). From the practical standpoint, the target HDL implementation can be modelled using intermediate formats, such as finite state machines, based on which a series of formulas can be constructed by a proof generator. Then a theorem prover determines the correctness of the formulas (Smith, 1997). While formal methods have evolved significantly over the past two decades, due to the state space explosion problem, they are still limited to small designs, or to small sub-blocks of large designs. Consequently, formal methods on their own are insufficient to prove the correctness of the entire design and their usage is enhanced by simulation-based methods, in particular constrained-random verification.

1.2.3 Physical verification

The layout representation of a design contains the information that is mapped from logical structures (logic gates, flip-flops) to the physical elements (transistors, wires). Physical verification checks the consistency between the logical design and the physical schematic. For example, timing analysis is focused on verifying whether the delays due to signal interactions are within the bounds estimated at the previous design steps (Sivaraman and Strojwas, 2012). The layout versus schematic (LVS) checking verifies the connection among physical elements according to the logical structure. The design rules checking (DRC) and electrical rules checking (ERC) verify if geometrical distances needed for correct fabrication are respected (Weste and Harris, 2011).

1.3 Manufacturing test

Manufacturing test, which takes place during the the post-silicon phase, checks if each fabricated device matches the design implementation. It is focused on screening for defects introduced during the fabrication process, e.g., shorts or open wires.

Over the years many fault models have been proposed to map the defect space, which is difficult to quantify, to a fault space which, for logic circuits, can be measured practically. In particular, static fault models, such as the stuck-at fault model, or the timing fault models, including transition or path delay faults, are widely employed to assess the effectiveness of manufacturing test. Despite the fact that no single fault model can reflect all the potential defects in a circuit, it is has been shown over the years that systematic adaptation of fault models has significantly improved the quality of manufacturing test (Wang *et al.*, 2006). Due to the fact that fault models correlate to the defect mechanisms and fault coverage can be used to estimate the quality of manufacturing test process, the field of manufacturing test has evolved as a scientific discipline over the past five decades. Systematic methods, such as algorithms for automatic test pattern generation (ATPG) have been developed, whose effectiveness is measured based on the fault coverage that can be attained.

Test generation, application and observation are difficult to handle, and they can be even practically infeasible, when the number of state elements (flip-flops or latches) becomes excessively high. Since VLSI circuits can have millions of state elements, design for test (DFT) methods are extensively used in practice, thus controlling and observing state elements. For example, scan chains configure state elements into shift register-like structures during test. Also built-in self-test (BIST) places pattern generators and response analyzers on the same die as the design-under-test (DUT)

(Stroud, 2002). The BIST method consumes additional logic resources, and hence on-chip area, however it can be especially effective when the tests need to be carried out in-field. Another DFT method is boundary scan, which wraps and isolates the DUT for better controllability and observability of the inputs and outputs (I/Os) when testing the interconnects on the printed circuit board (PCB). By controlling the test access ports (TAPs), according to the interfaces defined by JTAG (Joint Test Action Group) (IEE, 2013b), boundary scan can also be used for test access during logic test of components.

Both pre-silicon verification and manufacturing test have evolved in the last few decades into engineering disciplines with strong theoretical foundations. Post-silicon validation, which is concerned with confirming that no design errors have escaped the pre-silicon verification step, has been traditionally viewed as a necessary step of practical relevance. Many case studies have been discussed over the years by practitioners, however only at the turn of this century the need for systematic approaches was articulated (Vermeulen *et al.*, 2002). Considering that practical needs for post-silicon validation have existed since the early days of electronics, the relevance of post-silicon validation is highlighted in the next section.

1.4 Post-silicon validation

Pre-silicon verification is concerned with identifying and fixing design errors and therefore the RTL description is iteratively refined. The simulation-based verification finds design errors based on a large set of use cases. Nonetheless, simulation is known to be slow; for example, the study on a commercial microprocessor (Bentley, 2001) argues that it may take weeks of simulation of test cases that will take merely seconds to minutes of real-time execution. The inherent limitation of formal verification in modelling the whole design confines its applicability within focused units on small scales. Furthermore, when accounting for unique electrical states, such as the ones caused by process variations or effects exercised only under certain process-voltage-temperature corners, it becomes more difficult to develop both accurate and scalable pre-silicon verification methods (Mitra *et al.*, 2010).

Once the confidence level of pre-silicon verification is deemed to be sufficient, the implemented design is sent for fabrication. The main concern is that the confidence level, which is measured using metrics such code or state coverage, is often traded-off against verification time. Manufacturing test, on the other hand, is not concerned with finding and identifying subtle design errors (or bugs) that have escaped to silicon prototypes. After screening for manufacturing defects, the design is validated on a system platform. It is in this phase that subtle design errors (which affect every single fabricated device) are commonly uncovered. In order to compensate for the insufficiency of pre-silicon verification methods, the role of verification employed during the pre-silicon phase has to be continued on silicon prototypes. This critical step for finding design errors before committing to high-volume manufacturing, is commonly referred to as post-silicon validation (Keshava *et al.*, 2010).

The key benefit of post-silicon validation is that, unlike pre-silicon verification, which is commonly six to nine order slower than the real-time execution, the design-under-validation (DUV) can be stressed over extensive periods of time to reveal subtle errors that have escaped to silicon prototypes. In addition to known application use-cases, constrained-random sequences can be applied to the design during this phase. The main advantage of such randomized/functional tests is the huge volume of clock cycles, and in a few seconds of real-time execution more stimuli are applied to the DUV than during the entire pre-silicon phase. Hence a few hours, or possibly days or weeks (Adir *et al.*, 2011) of running validation suites on silicon prototypes can uncover (or help increase the confidence of the lack of) design errors.

Post-silicon validation lacks the controllability and the observability, which is taken for granted during pre-silicon verification. A large volume of proper stimuli are required for supporting the extensive periods during validation. The simulation tool in pre-silicon verification utilizes the constrained-random number generator to produce stimuli that satisfy user-defined constraints, whereas it does not fit the unique environment of the post-silicon phase. For instance, transmitting the stimuli generated from the simulation tool to the silicon prototype is impractical due to bandwidth limitations; meanwhile the volume of generated stimuli is limited by the slow execution of software tools. Considering validating a design with 128-bit inputs with the real-time frequency of 1GHz, running a validation session for one day will require at least 1 petabyte (10^{15} bytes) of stimuli. Consequently, one has to consider how to generate a large volume of randomized functional sequences on-the-fly. The attempt of emulating the behaviour of software algorithms for constrained-random stimuli generation on hardware is impractical with the exception of microprocessor-based designs where the microprocessor itself is used for stimuli preparation, application and observation.

The limited observability of the fabricated devices confines the methods of checking responses during post-silicon validation. In order to observe the erroneous response for further debug, many structured methods have emerged to improve the observability during validation, such as the in-system assertion checking, as well as the embedded trace buffers for recording the execution history after the erroneous behaviour occurs.

This dissertation is focused on enhancing the controllablity for post-silicon validation. A more detailed elaboration of the research background and related methods will be discussed in Chapter 2.

1.5 The contribution of this dissertation

Motivated by the need to improve the efficiency of compliant stimuli generation for post-silicon validation, and to reduce the hardware cost and the amount of data that is stored on-chip, the main contribution of this dissertation is a new programmable solution for on-chip constrained-random stimuli generation for post-silicon validation.

The scope of the proposed work is highlighted is shown in Figure 1.2. During pre-silicon verification, a software simulator can wrap the target design using testbench writing in an HDL to produce stimuli according to user specification. In order to facilitate controllable experiments during post-silicon validation, the proposed solution inserts constrained-random stimuli generator (CRSG) blocks during design-time, and a flow for iterative run-time configuration of these blocks (in Figure 1.2 the corresponding boxes are shown in gray and the steps will be detailed in the following chapters). In order to configure CRSG blocks, the user-defined constraints are converted into an intermediate form as binary cubes. At runtime, the cubes are loaded on chip to program the on-chip generator which generates random, yet functionally-compliant stimuli, according to the cubes with high throughput. A key advantage of configuration through user-programmability at runtime is the ability to control experiments and bias the constrained-random sequences as the validation progresses.

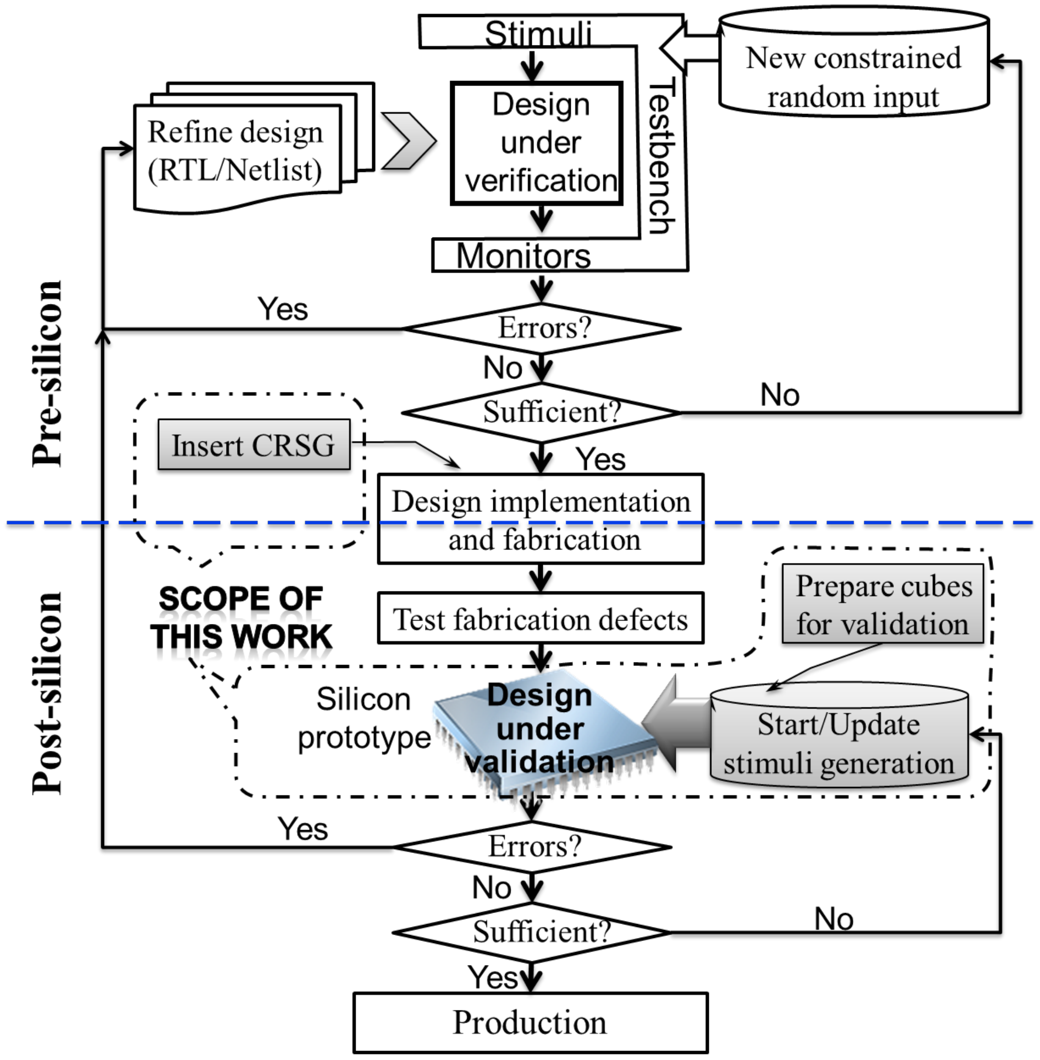


Figure 1.2: The scope of the work from this thesis during the implementation cycle.

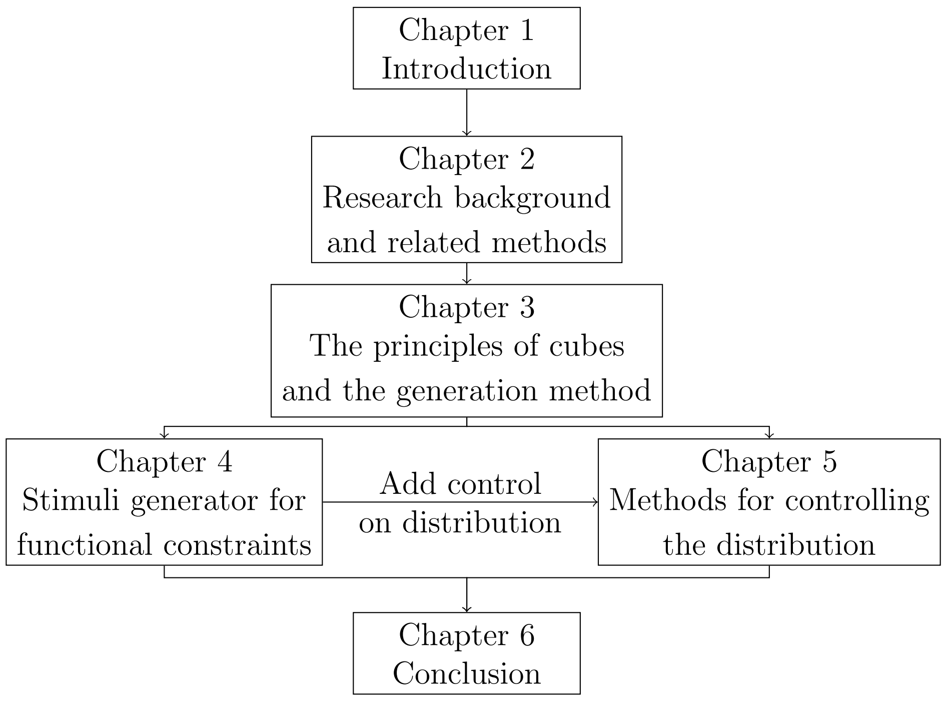

Chapter 1
Introduction
Chapter 2
Research background and related methods
Chapter 3
The principles of cubes and the generation method
Chapter 4
Stimuli generator for functional constraints
Add control on distribution
Chapter 5
Methods for controlling the distribution
Chapter 6
Conclusion

Figure 1.3: The relationship between the chapters from this dissertation.

1.6 The structure of the dissertation

The relation among the contents in the chapters is outlined in Figure 1.3. Chapter 1 has provided a general view of the methods for ensuring the quality ICs: pre-silicon verification, manufacturing test and post-silicon validation. Chapter 2 elaborates in more detail on the research and technical background, followed by the framework of the proposed solution that consists of two phases. The phase of cube generation is elaborated in Chapter 3. Regarding the phase of on-chip generation, Chapter 4 describes the on-chip generator structure for supporting SystemVerilog constraints on functionality, including logic and sequential constraints. The methods that extend the work to support the distribution of the generated stimuli are given in Chapter 5, including a method for uniformly-distributed stimuli generation followed by a method for customized distributions. The dissertation is concluded in Chapter 6.

Chapter 2

Background and related work

The practical approaches to both pre-silicon verification and manufacturing test have matured over the years and there are known good practices based on tool flows and algorithmic methods. Post-silicon validation has traditionally relied on ad-hoc methods, and automation has been used only within specific application domains, such as validation of microprocessor-based designs. There is little information in the public domain on systematic approaches used to validate generic logic blocks. The few methods that have been discussed in the public domain have been focused more on the observability aspects.

This chapter reviews the relevant approaches used at both the pre-silicon and the post-silicon stages for improving the controllability and observability during validation. Since the main objective of the proposed work is to enable the constrained-random methodology at the post-silicon stage, this chapter also examines the fundamentals of this methodology. The reader is provided with case studies that illustrate the basic usage of constraints during pre-silicon, and these examples serve as the motivation for the contributions presented in the latter chapters.

2.1 A brief review of pre-silicon verification methodologies

The concept of controllability and observability is conceptually derived from the theory of control systems (Gopal, 1993), and it has been widely used in the field of circuit verification. Essentially, controllability is concerned with driving the system into a specific state space by controlling the inputs. The observability is concerned with measuring the in-system states (and the transitions between them) at the observable outputs. This section provides an overview of the methods used for improving controllability and observability during simulation.

2.1.1 Controlling the state of the circuit

As the prevalent method during the pre-silicon stage, simulation-based verification applies a large set of use cases, in order to check the consistency between the specification and the design. A use case employs a series of stimuli that is applied to the targeted design to drive the design into specific functional states (e.g., set the overflow flag of an arithmetic unit). The stimuli can be in the form of executable programs for microprocessors, valid data packets for interface units, or a series of samples for a digital signal processing block. The stimuli must be valid inputs for the design, which need to satisfy of the design's interface protocols, and possibly ranges for values for different fields according to a specific verification scenario.

The stimuli can be manually written as direct verification use cases, which are useful for checking the known hard-to-reach states. Random verification, can automatically generate stimuli for the target design and help uncover unforeseen problems

(Yuan *et al.*, 2010). Some specialized randomization tools have been developed for microprocessors, e.g., Genesys-Pro for Power PC-based devices (Adir *et al.*, 2004). The randomization methods for generic circuits offer special features in hardware description languages (HDLs) to specify the constraints. During simulation, the constrained-random number generator embedded in the simulator generates stimuli that satisfy these user-defined constraints (Wiemann, 2007; Spear and Tumbush, 2012). Many widely used simulation tools, e.g., VCS (Synopsys, 2015) and ModelSim (Mentor Graphics, 2015), have integrated constrained-random generation algorithms for automatically generating stimuli according to user-specified requirements. The details of how the requirements are specified will be elaborated in Section 2.3.

2.1.2 Observing the state of the circuit

In addition to stressing the targeted design, it is critical to check whether the responses are consistent with what is expected according to the specification. Traditionally, the response from the behavioural reference model is used as the golden response, which is compared against the output from the simulated design. However, this approach is limited by the lack of sufficient details in the reference model. For example, although the reference model and the design that is verified are expected to have the same outputs, the internal details, such as intermediate states, might be inconsistent. If the software model is refined to compute more of this type of intermediate information, it will gradually become slower, thus becoming less practically feasible to compute golden responses. The same concern of lack of internal details applies if the reference model is a physical device. What are also difficult to verify using reference models are the features used to trade-off performance versus power.

Over the past decade or so, verification methods using assertion checking have become prevalent (Cerny *et al.*, 2014). Assertions do not necessarily relate to a specific implementation, since they are properties based on the specification that must hold regardless of the implementation. During simulation, assertion conditions are checked whether they are satisfied (`true`) or violated (`false`). They are particularly useful when using constrained-random inputs when the golden response is unavailable (Foster *et al.*, 2012). If no properties are violated for extensively long randomized functionally-compliant stimuli, the confidence level can be increased before sending the design for manufacturing.

2.2 State-of-the-art for post-silicon stage

Considering its unique environment, and the limited controllability and observability at the post-silicon stage, many approaches have been proposed to bridge the gap between pre-silicon to post-silicon validation (Nahir *et al.*, 2010). This section will provide an overview of the methods for stimuli generation (controllability), error detection (observability), as well as root causing (post-processing the failing information).

2.2.1 Stimuli generation

Similar to the methods at the pre-silicon stage, the direct generation involves manual development to exercise specific functions. The generation is closely coupled with the target design and leads to poor reusability. Random generation, on the contrary, compensates for this inflexibility and has been widely adapted in practice. A large volume of *random, yet functionally-compliant, sequences* are needed for exposing the design errors, which have escaped to the silicon prototypes (Nahir *et al.*, 2010; Mitra *et al.*, 2010; Sadasivam *et al.*, 2012; Adir *et al.*, 2011; Nicolici, 2012).

Considering that transmitting the constrained-random stimuli from simulation environments to the silicon prototype is obviously impractical due to bandwidth limitations, one has to consider how to generate a large volume of randomized functional sequences in real-time. For microprocessor designs, instruction-level templates (Sadasivam *et al.*, 2012; Adir *et al.*, 2011) are used to guide the on-chip random stimuli generation. The predefined templates fix the format and the instruction sequence, leaving the free fields open to randomization. The on-chip generation process can produce instruction sequences similar to the ones during simulation. The method in (Mitra *et al.*, 2010) proposed to generate executable machine code for generating the stimuli that stress the blocks under validation. These methods have been proven to be successful for the sub-modules within a microprocessor, as well as for the modules that are on the path between the microprocessor and the memory where the machine code is stored (e.g., cache controllers). However, it is not clear how they can be adaptable to hardware modules that are not directly accessible by the microprocessors. This is because many hardware accelerators (e.g., for video/networking) are often only configured by microprocessors, and the data they consume/produce is passed directly from/to the inputs/outputs (I/Os). In such cases, high-throughput signal generators are placed on-chip for the validation phase, however they are often customized to the specific needs of the design at hand, e.g., (Wu *et al.*, 2011). Therefore, for logic blocks and data channels not easily accessible or not controlled by programmable embedded microprocessors, high-throughput constrained-random signal generators placed on-chip near the target modules are expected to be employed to generate at-speed functionally-compliant stimuli on the silicon prototypes, which should be parameterized at design time and programmed in-system during validation time. This type of stimuli should be easily subjected to constraints, consistent with the specification and format of data packets fed to the design-under-validation in an application environment. A systematic way of designing such general constrained-random signal generators (CRSGs) is an active area of research (Nicolici, 2012), as elaborated below.

Some well-understood logic blocks can be employed at the core of CRSGs. The k-bit maximum-length Linear Feedback Shift Register (LFSR) generates $2^k - 1$ patterns if the characteristic polynomial is primitive and irredundant (Bardell *et al.*, 1987). The use of LFSR for compressed deterministic test has been introduced in (Köenemann, 1991) and this concept of reseeding LFSR has been refined and widely adopted in practice during the subsequent decade (Barnhart *et al.*, 2001; Rajski *et al.*, 2002; Wohl *et al.*, 2003). Also, many variants of the LFSR, e.g., de Bruijn counter, weighted pattern generator, and cellular automata (Wang *et al.*, 2006), have been proposed to control the pseudo-random stimuli sequences. Furthermore, there are known methods to alter pseudo-random sequences for manufacturing test, e.g., (Gherman *et al.*, 2004; Touba and McCluskey, 2001). Nevertheless, none the above-mentioned methods have been tuned to force all the pseudo-random stimuli to the unique functional constraints as defined in pre-silicon verification environments.

The methods in (Kinsman *et al.*, 2012, 2013) proposed to generate functionally-constrained pseudo-random sequences by removing the non-compliant stimuli by reseeding LFSRs. The details of such methods will be elaborated in Section 2.3. The reseeding-based method is one way of satisfying constraints at the post-silicon stage by transforming the autonomous sequence from the LFSR. Section 2.3 will also examine the constrained-random methodology at the pre-silicon stage and its adaptation to the post-silicon stage.

2.2.2 Error detection

Considering the limited observability in silicon prototypes, a series of approaches have been developed to improve the efficiency of error detection. An early error detection would benefit the work of isolating the errors, as well as finding out the root cause.

Early methods propose to reuse design-for-test (DFT) structures, e.g., scan chains, to dump the inner states that are hard to be observed for off-chip analysis. The method in (Vermeulen *et al.*, 2002) takes advantage of test access port of JTAG (IEE, 2013b) to control the scan chains for offloading the inner states. Wrapper registers have also been explored (Abramovici, 2008). Nonetheless, these methods start to dump the data when a wrong behaviour due to errors results in an observable failure, e.g., the halt of the system. Therefore, the latency from the time the error has been sensitized to the time the state is dumped, makes them unsuitable for design errors that manifest themselves after a long period of operation (Tang and Xu, 2008). This is because the scan dumps do not provide a history of events of interest that lead to the corrupted state. Beside, the scale of validating VLSI circuits necessitates a much higher controllability and observability beyond what manufacturing test needs

(Hopkins and McDonald-Maier, 2006).

Thus, some validation-specific structures for in-system monitoring have been investigated over the past decade. A group of pre-selected logic elements are observed in real-time according to a set of properties derived from the functional specification. On-chip trigger units (Ko and Nicolici, 2009) or hardware assertion checkers (Boule *et al.*, 2007) are employed to reduce the latency from error excitation to its detection. They are commonly used together with trace memories (Anis and Nicolici, 2007; Ko *et al.*, 2008) or footprint recorders (Park *et al.*, 2009), which can track a subset of relevant signals over a window that lead to the failure detection. It is important to emphasize that this type of methods for improving observability are critical for a constrained-random methodology where no golden responses are available. In practice, all the new methods discussed in the following chapters from this thesis, which are focused on improving the controllability during post-silicon validation, must be complemented by the methods used for providing real-time observability, such as the methods mentioned above.

2.2.3 Root cause analysis

At both pre-silicon and post-silicon stages, identifying an erroneous behaviour is followed by finding its root cause and fixing it (Wagner and Bertacco, 2010). When the errors are detected by response comparison or in-system monitoring logic, the stimuli that lead to the erroneous behaviour as well as the context from the trace logic are collected to diagnose the errors. By the iterative process of reproducing and analyzing the suspicious behaviour and conditions, the errors are gradually localized and isolated. In the meantime, the pre-silicon verification techniques could be used to facilitate error diagnosis. For instance, pre-silicon simulation can be used to replay the trace related to the erroneous behaviour (Chang *et al.*, 2007). Formal methods could also help identify the error condition (Gharehbaghi and Fujita, 2011). While root cause analysis is an important step, it is beyond the scope of the work from this thesis and the interested reader can refer to (Chang *et al.*, 2007; Wagner and Bertacco, 2010; Gharehbaghi and Fujita, 2011) for further details.

2.3 Describing constraints with equivalent models

So far the previous sections have summarized the relevant methods for improving controllability and observability during both pre-silicon verification and post-silicon validation. Since the scope of the proposed work is to facilitate the reuse of constraints from the pre-silicon to the post-silicon environment, this section provides the technical background for describing constraints and how they are modelled by different algorithmic approaches. It explains how constraints can be formalized in a verification language, such as SystemVerilog, and it also provides an overview of hardware-oriented processing of constraints.

2.3.1 Constraint solving for pre-silicon verification

Given a set of user-defined constraints for verification, the constraint solving engines in a simulator would find a group of value assignments to the variables that satisfy all the constraints; otherwise it is expected to report non-existence of such assignments and the reason for it (Yuan *et al.*, 2010). For instance, Table 2.1 enumerates all valid assignments according to the constraint $x \geq y$, in which x and y are 2-bit

Table 2.1: Valid assignments according to the constraint $x \geq y$, in which x and y are 2-bit unsigned integers.

x		**y**		**Digital pair**
x_1	x_0	y_1	y_0	
0	0	0	0	$\langle 0, 0 \rangle$
0	1	0	0	$\langle 1, 0 \rangle$
0	1	0	1	$\langle 1, 1 \rangle$
1	0	0	0	$\langle 2, 0 \rangle$
1	0	0	1	$\langle 2, 1 \rangle$
1	0	1	0	$\langle 2, 2 \rangle$
1	1	0	0	$\langle 3, 0 \rangle$
1	1	0	1	$\langle 3, 1 \rangle$
1	1	1	0	$\langle 3, 2 \rangle$
1	1	1	1	$\langle 3, 3 \rangle$

unsigned integers. As shown in the table, multi-bit, or word-level, variables are treated as a set of bitwise variables. The user-defined constraints are rewritten based on the bitwise variables, followed by analyzing and modelling them by using different methods described in this section. Some methods based on word-level constraints solving for high-level modelling have been proposed (Jaffar and Maher, 1994), however bit-level manipulation is eventually required whenever logic conditions arise. In the following, the typical models for constraint solving are examined.

SAT-based modelling

The Boolean satisfiability (SAT)-based approach offers a search-based way of constraint solving via modelling Boolean expressions. Given a Boolean expression in the conjunctive normal form (CNF), a SAT solver finds the assignment to the Boolean variables such that the Boolean expression evaluates to `true`. A CNF expression is a conjunction (joined by Boolean `AND`) of clauses, where each clause is a disjunction (joined by Boolean `OR`) of literals (a Boolean variable or its negation). Taking a CNF expression $(a + b + c)(b + d)(a + \bar{d})$ as an example, the three clauses are $a + b + c$, $b + d$ and $a + \bar{d}$.

The fundamental idea of SAT-based constraint solving is to convert the user-defined constraints into a CNF formula, based on which the valid assignments are searched by a SAT solver. For instance, the previous constraint $x \geq y$ can be expressed as $(x_1 + \bar{y_1})(x_0 + \bar{y_1} + \bar{y_0})(x_1 + x_0 + \bar{y_0})$. It can be verified that only the assignments in Table 2.1 can make the expression `true`. Solving a SAT problem is known to be NP-complete (Cook, 1971), nevertheless, many SAT algorithms have been developed that have been proven to be efficient in practical applications. The early DPLL algorithm (Davis *et al.*, 1962) iteratively assigns variables and analyzes conflicts in clauses until the valid assignments are found (or a proof of unsatisfiability is given by identifying the conflicting clauses). Modern SAT solvers adapt a series of refined algorithms to enhance the efficiency in the search process, e.g., conflict-based learning (Marques Silva and Sakallah, 1996), or watched literals and decision heuristics (Moskewicz *et al.*, 2001). The processes of constraint solving using CNF and similar Boolean forms have been proposed for functional stimuli generation for pre-silicon verification (Fallah *et al.*, 2001; Kitchen and Kuehlmann, 2007).

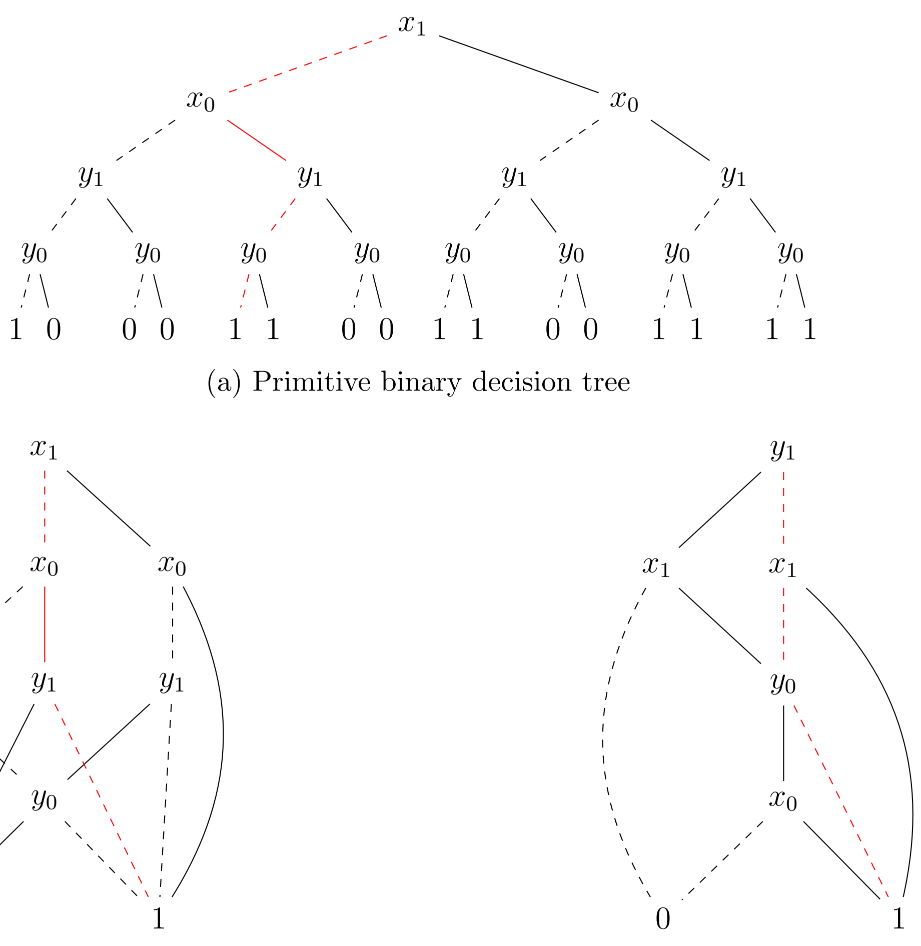
BDD-based modelling

Graph-based approaches using binary decision diagrams (BDDs) have also been proposed to transform the constraints for finding valid stimuli. BDDs were introduced in

(Lee, 1959; Akers, 1978) and regularized in canonical form in (Bryant, 1986). BDDs are directed acyclic graphs to represent the truth table for a given Boolean formula. The inner nodes represent variables in the formula. A BDD includes two types of edges, indicating if the variable of the node is assigned to 1/`true` or 0/`false` respectively. The only two terminal nodes of 0 and 1 indicate whether the assignments on a path ending with this terminal make the formula `false` or `true`. BDDs have been widely adopted in many design and verification steps, such as model checking (Burch *et al.*, 1990), circuit test and optimization (Cho *et al.*, 1993), and constraint solving for simulation (Yuan *et al.*, 2004).

Figure 2.1(a) illustrates the primitive binary decision tree for the previous constraint $x \geq y$. Each path from the root node to a leaf node is mapped to one assignment in the truth table. If the leaf is 1, it indicates the assignment satisfies the constraint. It can be verified that all the paths ending with leaf 1 cover the valid assignments as shown in Table 2.1. A tree-based structure expands exponentially to the number of variables in the formula, (e.g., Figure 2.1(a) contains $2^4 - 1$ inner nodes) which becomes practically infeasible for complex cases. The essence of a BDD is to reduce the number of inner nodes by merging nodes and reordering variables, as exemplified in Figure 2.1(b) and (c) respectively. Although a reduced ordered BDD can reduce the number of nodes significantly when compared to a binary decision tree, it is sensitive to the variable order and, in the worst case, it still suffers from exponential memory complexity.

Some efficient BDD packages for BDD construction and manipulation are available in the public domain (Somenzi, 2012). By using a BDD package, the constraints can be modelled as BDDs, based on which the constrained stimuli can be generated via



(a) Primitive binary decision tree

(b) Reduce nodes

(c) Reorder variables

Figure 2.1: The sketch from a tree to BDD for the constraint $x \geq y$. The solid edge indicates its parent node is assigned to 1; otherwise 0 if it is a dashed edge. For instance, the red path in (a) indicates $x_1x_0y_1y_0 = 0100$, which is then mapped to the equivalent red paths during the reduction and reordering process shown in (b) and (c).

searching for the compliant paths.

Modelling by Boolean unification

The Boolean constraints can be resolved into a vector of Boolean functions (Yuan *et al.*, 2010), which is based on the classical algorithms of Boolean unification (Büttner, 1988; Martin and Nipkow, 1989) and has been adopted in parametric Boolean equation solving (Fujita *et al.*, 1991; Aagaard *et al.*, 1999).

This approach considers the constraints as the compulsory relations among the variables, which makes the variables dependent on each other. The essence is to express each bitwise dependent variable in the constraint by a Boolean function of some independent variables, denoted as parametric variables. Finally, the parametric variables are assigned with arbitrary values, based on which the compliant stimuli can be calculated. For instance, the constraint $x \geq y$ is a Boolean function of four bitwise variables x_1, x_0, y_1, y_0 as illustrated in Table 2.1. According to the constraint, the four variables are expressed by corresponding four Boolean functions of four free parametric variables a, b, c, d as shown in Equation (2.1). Table 2.2 enumerates all the possible assignments for parametric variables and the corresponding stimuli. The stimuli are the same as the ones from Table 2.1.

$$\begin{cases} x_1 & = a \\ x_0 & = b \\ y_1 & = ac \\ y_0 & = a\bar{c}d + bd \end{cases} \tag{2.1}$$

Table 2.2: Calculated stimuli according to the Boolean unification result for the constraint $x \geq y$.

Parametric variables				x		y	
a	b	c	d	x_1	x_0	y_1	y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1

2.3.2 Capturing constraints using SystemVerilog

SystemVerilog is a widely adopted hardware description language (HDL), which can be used for hardware description at the register-transfer level (RTL) and the gate level, as well as hardware verification. In a sense, SystemVerilog serves as an unified combination of Verilog (IEE, 2006), a prevalent hardware description language, and the verification methodology (Bergeron *et al.*, 2006). The typical features for verification include the constraints description, the support for coverage and abstracted data structures and control of the simulation events (Spear and Tumbush, 2012). In particular, it standardizes the constraint formats and their behaviour using a variety of programming features and libraries. In the following, the features related to constrained-random value generation in the up-to-date SystemVerilog standard (IEE, 2013a) are examined. Although the features in the standard are aimed at pre-silicon verification, they could be used as the objectives for the solutions developed for post-silicon validation.

The random variables can be declared using **rand** and **randc** type-modifier keywords. The difference is that, the **rand** variables are standard random variables which are uniformly distributed over the valid range; meanwhile the **randc** variables are random-cyclic variables which randomly iterate over all the values in the valid range without repetition within an iteration. The support of random-cyclic distribution may consume more computing resources (i.e., CPU time and memory) for avoiding repetition. Considering the cost for supporting **randc**, the standard tolerates the limit of the length for a **randc** variable to be imposed by simulators (so long as it is not less 8 bits).

Constraints can be expressed using constraint blocks. The block supports most functional operators, as used in hardware design, including arithmetic operators (`+`, `-`), shift operators (arithmetic/logic shift left/right), logic operators (`&`, `|`, `!`, `^`), relation operators (`>`, `>=`, `<`, `<=`, `==`). The implication constraint can be defined using the **if-else** statement. Code 2.1 shows an example of a constraint, which is encapsulated in a **class** block.

Code 2.1 A SystemVerilog class with a constraint for two 2-bit variables.

```
class GreaterEqual;
  rand bit[1:0] x, y;
  constraint good { x>=y; }
endclass
```

Code 2.2 The SystemVerilog constraint for weighted distribution.

```
class DistConstraint;
  rand bit [15:0] x;

  constraint x_dist {
    x dist { [10:999]:/3, [1000:2000]:/2 };
  }
endclass
```

Apart from the functional constraints represented by a set of logic expressions, referred to as logic constraints , SystemVerilog offers the **randsequence** structure for defining a sequence of constraints, referred to as sequential constraints. Detailed examples for logic and sequential constraints are given in Section 2.4.

Regarding the issue on the control of distribution, the modifier **rand** and **randc** have defined uniformly-distributed sampling. Nonetheless, the user-defined constraints may impose bias on the distribution of valid stimuli. SystemVerilog offers the distribution operator **dist** to specify weighted distributions, which imposes sampling with more or less likelihood in the specific sub-range. For instance, Code 2.2 illustrates the constraint on the distribution of 8-bit unsigned variable **x**. It means **x** could be sampled in the ranges of [10, 999] and [1000, 2000] with the weighted ratio of 3:2. One can use **solve** and **before** to control the priority for variable evaluation (**randc** is not allowed with these expressions). Note, however, improper constraints on the solve order can result in failure to generate compliant stimuli.

The SystemVerilog language also offers dynamic constraint modification in the testbench during simulation, e.g., to enable/disable parts of constraints. This is useful for switching between different verification sessions. In addition, the standard also clarifies the peripheral issues related to object-oriented scheme (e.g., the scope rules, pre/post steps) and the stability of software generator (e.g., initialization, multi-threading).

In summary, the key points for constrained-random stimuli generation include the support for logic and sequential constraints, the control on distribution and dynamic programmability. Commercial simulators, such as VCS (Synopsys, 2015) and ModelSim (Mentor Graphics, 2015), are compliant to the standard and widely used in pre-silicon verification. A few case studies using SystemVerilog will be given in Section 2.4 in order to illustrate the types of features that can be supported by the methods described in the subsequent chapters.

2.3.3 Hardware-oriented representation for constraints

For pre-silicon verification, the simulation tools can take advantage of powerful computing resources on which they run and convert the user-defined constraints into the equivalent representations that were elaborated in the previous subsections. The post-silicon validation environment lacks such computing resources and therefore potential formats of on-chip representation for user-constraints are discussed next.

Hardware synthesis

According to SystemVerilog, the functional expressions used for constraints have similar formats as expressions for hardware design. Hence there is opportunity of adapting

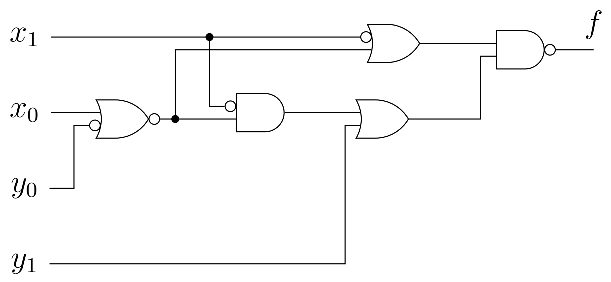


Figure 2.2: The synthesized hardware logic for Boolean expression $x[1:0] \geq y[1:0]$.

the technique of hardware synthesis to on-chip representation of constraints.

An intuitive way is to synthesize the constraints as a Boolean expression into hardware logic. It can be used to determine whether a given stimulus satisfies the constraints. For instance, Figure 2.2 illustrates the synthesized hardware logic for the constraint $x \geq y$. Given a primitive stimuli $\langle x, y\rangle$, the logic evaluates f to be `true` if and only if the constraint holds, so that it determines whether it should be a valid output or rejected. A trial and error approach is impractical if the potential compliant stimuli are sparse, in which case most stimuli would be rejected. Hence concurrent enumeration and refined sampling methods, such as Markov chain Monte Carlo (MCMC), have been adopted for hardware acceleration of constraint solving (Welp *et al.*, 2012). However, the hardware cost of such methods is prohibitively high in order to be place on-chip together with the design that is validated.

Alternatively, some intermediate representations used for constraint solving can be used to synthesize the constraint logic. For instance, Equation (2.1) from Boolean unification can be synthesized into hardware logic, which can directly compute the compliant stimuli without rejection, although many stimuli may be repeated as illustrated in the example from Table 2.2. Also the method from (Kukula and Shiple, 2000) can be used to implement hardware with multiplexors and `OR` gates based on a

BDD representation.

The key limitation of synthesizing constraints to hardware is the lack of in-system programmability. Once the hardware logic is fixed, according to the pre-defined constraints, it cannot be changed after fabrication. Since validation engineers do not know all the constraints that need to be used before tapeout, it is critical to have the in-system programmability feature for post-silicon validation. Therefore the approaches based on hardware synthesis are limited to applications based on field-programmable gate arrays (Wolf, 2004).

Programming LFSRs using seeds

As introduced in Section 2.2, LFSRs are traditionally used as on-chip random stimuli generators. However, they are no obvious ways how they can be controlled to generate outputs according to user-specified constraints. The methods from (Kinsman *et al.*, 2012, 2013) were the first to describe how to generate functionally-constrained pseudo-random sequences. These methods are based on reseeding LFSRs.

Consider the case of generating stimuli containing two 4-bit signals x and y, the valid stimuli are constrained as follows: $x \geq y$. As shown in Figure 2.3(a), the bare LFSR generates a sequence of random stimuli, among which only some stimuli are valid. Hence the reseeding logic is added to control the state of LFSR as shown in Figure 2.3(b). Before the LFSR generates an invalid stimulus, the pre-computed seed would be loaded into the LFSR, hence skipping the invalid subsequence. The preparation for the seeds requires solving systems of equations. The solvability and the frequency of reseeding depend on the LFSR configuration and constraints.

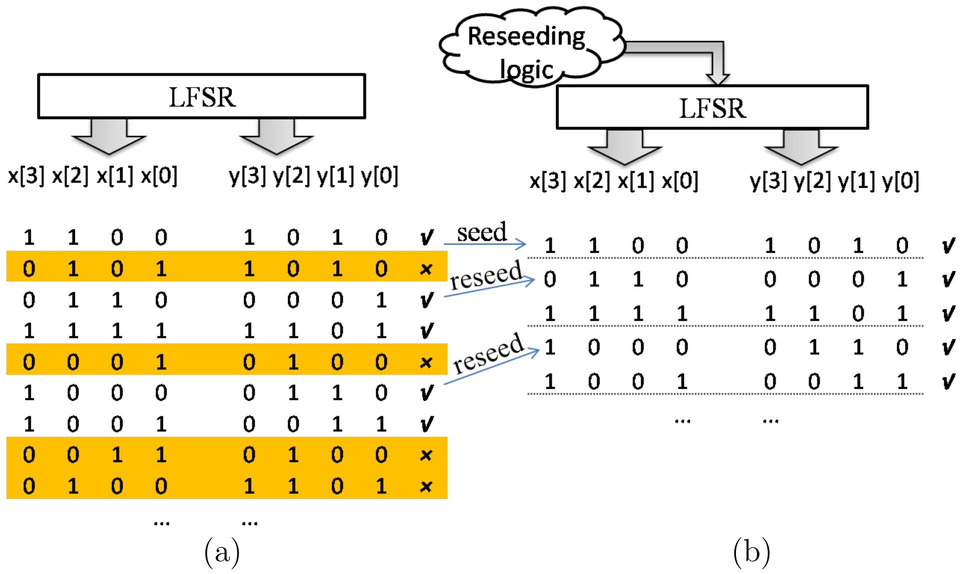


Figure 2.3: Constrained-random stimuli generation by employing reseeding logic circuitry around the LFSR. In order to force the output from the LFSR shown in (a) to satisfy the constraint $x \geq y$, the logic shown in (b) changes the state of LFSR with the new seed whenever a non-compliant output would be generated.

The LFSR reseeding method eliminates the necessity of creating hardwired circuitry, as the hardware synthesis-based methods require. Instead, it utilizes reseeding logic attached to the LFSR. In order to map the functionally-compliant sequences onto LFSR, the first step is to transform the compliant stimuli generated by the testbench during simulation into stimuli cubes. Considering the constraint $x \geq y$ (x and y are two 4-bit inputs) all the valid stimuli can be merged into *cubes* shown in Table 2.3. This can be achieved, for example, using an off-the-shelf logic minimization tool such as Espresso (McGeer *et al.*, 1993). Since the cube-based representation is also central to the work from this thesis, more details will be provided in Chapter 3.

The results from (Kinsman *et al.*, 2012, 2013) showed that it could generate a high volume of compliant stimuli using the seed data of a few kilobytes to megabytes. The generation of seeds requires computational resources and it might even happen that no solutions are found due to the structure of the LFSR and the linear dependencies

Table 2.3: Part of stimuli cubes according to the constraint `x[3:0]>=y[3:0]`.

x[3:0]	y[3:0]
1XXX	0XXX
11XX	X0XX
X1XX	00XX
11XX	XX00
XX1X	00X0
1X1X	X0X0
X11X	0XX0
111X	XXX0
XXX1	000X
1XX1	X00X
...	...

introduced by its feedback logic (especially if the number of specified bits in the cube is high).

Inspired by the cube-based method from (Kinsman *et al.*, 2012, 2013), the solutions from this dissertation explore an alternative approach, by directly using stimuli cubes to alter the sequences generated by LFSRs. In particular, the work from this thesis can be used to support both logic and sequential constraints, and, more importantly, it can be adapted to control the distribution of stimuli.

2.4 Case studies using SystemVerilog constraints

The key objective of the work from this dissertation is to provide an automated methodology to reuse constraints from pre-silicon verification in a post-silicon validation environment. In this section, a few case studies are provided that illustrate the basic features for constraint description that exist in SystemVerilog.

During the pre-silicon verification phase, the constraints are described in SystemVerilog (IEE, 2013a), based on the application-specific functionality and verification requirements. The syntax supports the functional constraints on variable values (e.g., Boolean expressions, or if-else and implication relations), constraints on distributions of patterns, and constraints on the solving order during simulation.

Functional constraints are divided into two broad categories: logic constraints and sequential constraints. Logic constraints capture the static features and the format of a pattern, while the sequential constraints capture the dynamic behavior during generation. For example, constantly generating positive 8-bit signed integers is considered as a logic constraint, e.g. $x \geq 0$, since it does not change from one clock cycle to another. Conversely, generating a positive integer and a negative integer in two adjacent clock cycles (i.e., the period $T = 2$) can be considered as a sequential constraint, because two logic constraints, i.e., $x \geq 0$ and $x \leq 0$, are needed in alternate clock cycles. One can consider logic constraints as a special case of sequential constraints (when $T = 1$).

Both logic constraints and sequential constraints play an important role in generating stimuli for validating digital hardware. In the following, typical use cases are presented and their characteristics are examined.

2.4.1 Logic constraints in the same clock cycle

Consider the functional verification of a simple microprocessor using randomly generated instructions. Code 2.3 shows an example using a SystemVerilog class, where different instructions have distinct constraints on the operand fields. Specifically, the

Code 2.3 A SystemVerilog class with logic constraints for generating ALU instructions.

```
typedef enum {ADD, SUB, SHIFT_L, SHIFT_AR, SHIFT_LR} op_type;
class StimuliForALU;
  rand op_type opcode;
  rand bit[7:0] opr1, opr2;
  constraint opr_range {
    if (opcode==SHIFT_L || opcode==SHIFT_AR ||
        opcode==SHIFT_LR) {
      opr2 inside {[0:7]};
    }
  }
endclass
```

second operand for shift left (**SHIFT_L**), shift arithmetic right (**SHIFT_AR**) and shift logic right (**SHIFT_LR**) instructions is constrained to be less than 8. The constraint captures the relationships between fields within the same clock cycle. Note that during stimuli generation there are no constraints in between two consecutive instructions.

Another practical example is the verification of the floating-point unit (FPU) in a microprocessor, which takes real numbers as inputs. On the one hand, each operand must be constrained based on the specified standard. Code 2.4 gives an example for generating an IEEE 754 standard compliant floating-point number (IEE, 2008). On the other hand, the FPU contains several logic parts for different arithmetic operations. In order to, for example, identify a potentially erroneous division calculation when the divisor and the dividend fall in a specific range, whenever the **opcode** is decoded as floating-point division, the constraints on the operand fields can be user-programmed such that the divisor and the dividend are within the ranges that stress the arithmetic blocks which were not sufficiently exercised before. The constraints on

Code 2.4 A SystemVerilog class for generating a single precision floating-point number compliant with IEEE 754 standard.

```
class FPNumber;
  rand bit sign;
  rand bit[7:0] exponent;
  rand bit [22:0] fraction;
  constraint number_range  {
    exponent = 127;
    fraction[22:21] inside {[0:2]};
  }
endclass
```

the operands can be flexibly updated during the verification process, so as to focus on the suspicious value ranges where the division calculation might be incorrectly implemented.

2.4.2 Sequential constraints over consecutive clock cycles

The randomization of instruction operands above was only concerned with the relationship between instruction fields, rather than constraints between consecutive instructions. Some designs and applications might require a series of constrained stimuli where the randomized pattern in each clock cycle follows a prescribed order and/or format based on specific protocols or standards.

An example of requiring sequential constraints can be found, for example, in network-on-a-chip communication scenarios. During a complete data transfer session, the sender module may issue a series of specific packets including data packets, as well as handshaking with the receiver module. Hence the field for identifying the type in each packet may vary from a session request packet, to a data packet, to a

Code 2.5 A SystemVerilog task capturing sequential constraints using a single constraint class for randomization.

```
logic [63:0] stimulus;
task InjectRandomSession;
  begin
    class GenerateSession session=new;
    session.randomize();
    stimulus={32'd0, session.SyncPacket};
    @(posedge clk);
    stimulus=session.DataPacket;
    @(posedge clk);
    stimulus={48'd0, session.FinishPacket};
  end
endtask
```

session end packet. In this case, the field for the packet type is constrained to enumerate the three values in three adjacent cycles. Code 2.5 illustrates the constraints using a SystemVerilog `task` block that resembles a protocol-based transmission session with 3 steps: a synchronization request, data transmission and the finish request. The class **GenerateSession** is assumed to contain 3 randomization members (**SyncPacket**[31:0], **DataPacket**[63:0] and **FinishPacket**[15:0]), which denote the 3 types of packets. Each of these 3 different types of packets can have their own relationships between fields and randomization needs, which can be captured as logic constraints. Note, the 3 packets may have different lengths, in which case the output stimulus must be sufficiently large to hold the longest packet.

While the random number generators embedded in pre-silicon environments are designed to generate stimuli consistent with the task block given in Code 2.5, it is not obvious how to port such features to hardware. Even if a mechanism is provided to alter the pseudorandom values produced by the LFSR in order to meet logic constraints, as it is the case for the previous methods (Kinsman *et al.*, 2012, 2013), it is also necessary that the sequential constraints can be applied at a rate (samples per clock cycle) as fast as during pre-silicon verification. When using sequential constraints is of key importance for post-silicon validation environments, it is critical that the hardware generator can handle constraint switching on a clock cycle basis. If the task block requires a constraint change in every clock cycle, the method presented later in Section 4.3 is capable of handling this feature.

As another example, consider a large frame that can be partitioned into smaller slices, which are then transmitted sequentially. Each of these slices has their own logic constraints that guide randomization. The task example in Code 2.6 uses **randsequence** block to generate one frame in multiple cycles, where 3 randomization classes are employed. The task block captures the sequential constraints to generate a complete packet where each of the slices (**head**, **tail** and multiple **body** slices) can have their own relationships between fields that guide randomization. As it was the case for the example from Code 2.5, the hardware generator is expected to apply different constraints to each slice. Even if these constraints need to be switched every clock cycle, the CRSG for sequential constraints presented in Section 4.3 can meet this objective.

2.5 Summary

This chapter has provided an overview of the relevant methods for improving the controllability and observability during both pre-silicon verification and post-silicon validation. It has also illustrated the technical background on how constraints are handled during pre-silicon verification and it has motivated the use of a cube-based

Code 2.6 A SystemVerilog task capturing sequential constraints using multiple constraint classes for randomization.

```
logic [15:0] stimulus;
task GenerateRandomFrame;
  parameter body_size=16;
  begin
    FrameHead head=new; // The class for the head
    FrameBody body=new; // The class for the bodies
    FrameTail tail=new; // The class for the tail
    randsequence (frame)
      frame : fhead repeat(body_size) fbody ftail;
      fhead : { head.randomize();
                stimulus=head.content;
                @(posedge clk); };
      fbody : { body.randomize();
                stimulus=body.content;
                @(posedge clk); };
      ftail : { tail.randomize();
                stimulus=tail.content;
                @(posedge clk); };
    endsequence
  end
endtask
```

representation for the proposed work. The key feature of the cube-based representation is that it can enable the in-system programmability of constraints. This chapter also provided a few case studies to illustrate some of the challenges faced when porting constraints from a pre-silicon to a post-silicon environment.

In the following chapters, Chapter 3 discusses the topic on cube-based representation of constraints in detail. Then Chapter 4 describes the proposed solutions for both logic and sequential constraints. The new method proposed for controlling the distribution of stimuli will be elaborated in Chapter 5.

Chapter 3

Representation of constraints as a set of cubes

As motivated in the previous chapter, due to the limited communication bandwidth between a post-silicon validation platform and the host where a simulator is running, it is impractical to reuse the stimuli that are generated by pre-silicon verification tools. Therefore, the valid space captured by the constraints described in a verification language, e.g., SystemVerilog, is translated into a set of cubes. This cube-based representation takes a central role in the methods presented in the following chapters. It serves as an efficient representation that captures the same information from the user-specified constraints, which can be transferred to the design-under-validation (DUV) for on-the-fly stimuli generation.

In this chapter the basic concepts of cubes are presented. Then the flow of generating cubes from the user-defined constraints is described, followed by the basic idea of how the cubes can be used for on-chip constrained-random stimuli generation.

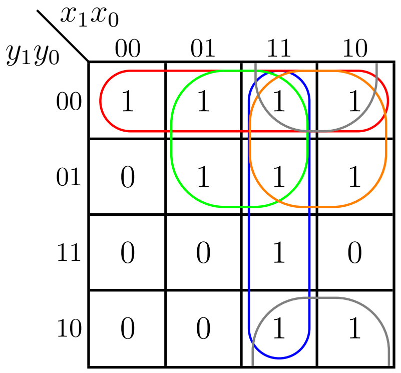
3.1 The concept of a cube

A cube is an m-bit vector comprised of '0's, '1's and 'X's (don't-care bits). Each cube is equivalent to a set of binary vectors where 'X's are replaced with either '0's or '1's. Given the whole set of m-bit binary vectors U_m, a cube is an equivalent symbol for a unique element in the power set of U_m (denoted as 2^{U_m}). Note, not every element from the power set can be expressed as a single cube. Each cube represents a regular super cubic sub-space in the m-dimensional binary vector space (that is why it is called a *cube*). For instance, the 4-bit cube "1X0X" is equivalent to the binary vector set {1000, 1001, 1100, 1101} as the implied space of this cube, which is also an element in 2^{U_4}. A permutation of values specified by a cube is a sequence that can be expanded from the respective cube.

Given a group of user-defined constraints, the equivalent set of cubes exactly covers all the *true* elements in the Karnaugh map (K-map) of the constraint function $f(s)$, in which the Boolean function $f(s)$ is evaluated as *true* if and only if the stimulus s satisfies all the constraints. For generating constrained-random stimuli, the proposed method chooses to convert the user-defined constraints into a set of cubes. Then the cubes are loaded onto on-chip, where the 'X's are substituted with random-filled bits, thus generating compliant stimuli. For example, given the constraint $x \geq y$ as shown in Code 2.1, the cube set of 4-bit stimuli (with the format as $x_1x_0y_1y_0$) is {1X0X, 11XX, XX00, X10X, 1XX0}. Each cube in the set denotes a value space, and the union of the five cubes covers the entire compliant value space for the constraint. The K-map shown in Figure 3.1 illustrates an intuitive example for the constraint function.

For the sake of correct generation of stimuli on-chip, the set of cubes converted

Figure 3.1: The K-map for Boolean function $x \geq y$, where x and y are 2-bit unsigned integers.



from the pre-silicon constraints should satisfy two fundamental properties.

Completeness: Any stimulus compliant to the constraints is able to be generated, or expanded from at least one cube in the set.

Sufficiency: Any stimulus expanded from a cube in the set must comply to the user-defined constraints.

The two properties need to be guaranteed by the conversion algorithm which generates the set of cubes from user-defined constraints. Further details on cube generation are provided in Section 3.3.

3.2 Formal notations for cubes and their characteristics

The use of ‘X’s distinguishes cubes from binary vector and makes the cubes efficient for representing constraints. In this section, the formal notations and characteristics for a cube, a cube pair and a set of cubes are given. The analysis for the cube pairs and for the set of cubes is particularly useful for the refinement of the distribution of generated stimuli, which will be presented in Chapter 5.

3.2.1 The characteristics of a cube

Given an m-bit cube $\boldsymbol{a}$ ($\boldsymbol{a} = \overrightarrow{a_{m-1}a_{m-2}\cdots a_0}$), the function for checking an 'X' bit is defined as:

$$\mathcal{X}(a_i) = \begin{cases} 1 & \text{if } a_i = X, (0 \le i < m) \\ 0 & \text{Otherwise, i.e., } a_i = 0 \text{ or } 1 \end{cases} \tag{3.1}$$

Then the total number of 'X's in an m-bit cube $\boldsymbol{a}$ is:

$$\xi_{\boldsymbol{a}} = \sum_{i=0}^{m-1} \mathcal{X}(a_i) \tag{3.2}$$

3.2.2 The characteristics of a cube pair

Given a pair of m-bit cubes $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$, ($\boldsymbol{a} = \overrightarrow{a_{m-1}a_{m-2}\cdots a_0}$ and $\boldsymbol{b} = \overrightarrow{b_{m-1}b_{m-2}\cdots b_0}$), the first cube $\boldsymbol{a}$ is called the reference cube. Each bit pair $\langle a_i, b_i \rangle$ in the same position i ($0 \le i < m$) is categorized as one of three cases:

- Mutually exclusive bit pair, if $\langle a_i, b_i \rangle \in \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\}$.
- Compatible bit pair, if $\langle a_i, b_i \rangle \in \{\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle X, X \rangle, \langle X, 0 \rangle, \langle X, 1 \rangle\}$.
- Overlapped bit pair, if $\langle a_i, b_i \rangle \in \{\langle 0, X \rangle, \langle 1, X \rangle\}$.

According to the types of bit pairs in the cube pair $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$, the properties can be deduced on the lattice $(2^{U_m}, \cup, \cap)$:

- If there is at least one mutually exclusive bit pair, then $\boldsymbol{a} \cap \boldsymbol{b} = \emptyset$. Also due to the commutativity of the mutually exclusive bit pair, $\boldsymbol{b} \cap \boldsymbol{a} = \emptyset$. Then $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$

and $\langle \boldsymbol{b}, \boldsymbol{a} \rangle$ are both mutually exclusive cube pairs.

- If all the bit pairs are compatible pairs, then $\boldsymbol{b}$ is a sub-cube of $\boldsymbol{a}$, i.e., $\boldsymbol{b} \subseteq \boldsymbol{a}$ or $\boldsymbol{b} \cap \boldsymbol{a} = \boldsymbol{b}$.
- Otherwise, i.e., there is no mutually exclusive bit pair, however there is at least one overlapped bit pair in them, then $\boldsymbol{b} \cap \boldsymbol{a} \neq \emptyset$ and it is defined as an overlapped cube pair.

For example, $\langle 00XX, 1XX0 \rangle$ is a mutually exclusive cube pair since the most significant bits in the two cubes (i.e., one is '0' and the other is '1') are mutually exclusive bit pairs. The cube "10X0" is a sub-cube of "1XX0". The cube pair $\langle XX00, 1XX0 \rangle$ is categorized as an overlapped cube pair. Intuitively, some elements of the cube "1XX0" overlap with some elements of the cube "XX00" in the K-map shown in Figure 3.1.

3.2.3 The characteristics of a set of cubes

For a set of m-bit cubes $s = \{\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \cdots\}$, the total number of cubes is $|s|$. The X-density of the ith variable (or position) is defined as :

$$\rho_i = \sum_{j=1}^{N} \mathcal{X}(a_i^{(j)})/|s| \tag{3.3}$$

Hence $\rho_i|s|$ indicates the number of unspecified bits at the ith position for all the cubes. Then the weight of a cube $\boldsymbol{a}$ is defined as:

$$w_{\boldsymbol{a}} = \sum_{i=0}^{m-1} \left[W_X \mathcal{X}(a_i) + \rho_i \mathcal{X}(a_i) \right] \tag{3.4}$$

The constant W_X is a tuning constant which denotes the weight of an 'X' bit. The first component in Equation (3.4) accounts for the size of implied vector spaces, i.e., the number of binary vectors which the cube can imply. Note, the actual size of the space is 2^{ξ_a}, however the logarithm of the entire space is used in the equation to match the range of the second component. The second component measures the correlation with other cubes in the set, according to the number of overlapped 'X' bits in the same position. Note, since W_X can be tuned by the user, for example by setting W_X to m it is guaranteed that a cube with more 'X' bits always has a larger weight than one with fewer 'X' bits.

3.3 Converting constraints to cubes

The user-defined constraints specified in SystemVerilog need to be converted into a cube-based representation. The general flow in the cube processing phase is shown in Figure 3.2. The SystemVerilog constraints are converted into an equivalent set of cubes. Then, as discussed in detail in Chapter 4, the cubes are mapped into binary cubes and they can be further encoded into compact binary cubes (CBCs) to reduce the requirements for on-chip storage.

Concerning the conversion of constraints written in SystemVerilog to a set of cubes, two approaches are discussed in this chapter. The first approach investigates the specific attributes of operators used in constraints, based on which it can generate the set of cubes by iterative enumeration and merging. The second approach combines the traditional techniques of hardware synthesis and BDD-based generation of cubes. Both approaches will generate a set of cubes that can be further reduced using any third party two-level logic minimization tool, e.g., Espresso(McGeer *et al.*, 1993).

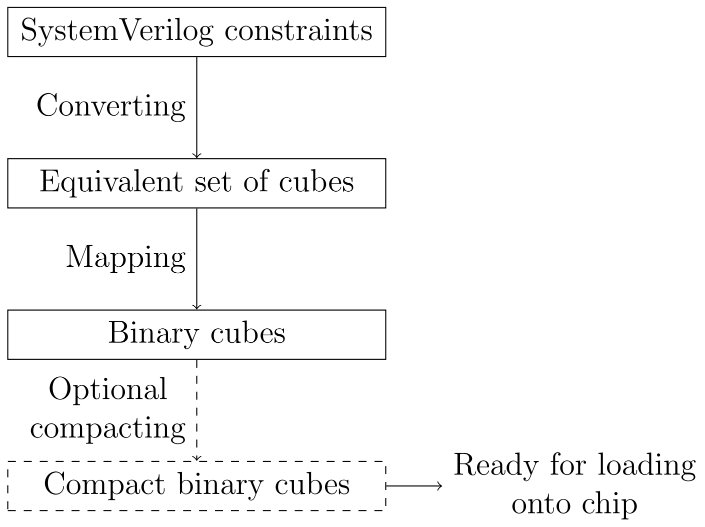


Figure 3.2: The general flow of cube processing.

3.3.1 Converting constraints to cubes using customized algorithms

The essence of conversion from constraints to cubes is the same as Boolean function minimization. According to the constraint operators, a group of algorithms can be designed, which iteratively enumerate cubes from constraints. A typical iterative flow of the algorithm is elaborated in the following.

The K-map shown in Figure 3.1 illustrates an intuitive example for the constraint $x \geq y$ in which x and y are unsigned 2-bit variables. It enumerates all the compliant pairs and fills '1's in the corresponding boxes. The objective of cube generation is to find the minimum logic expression. In this case, the result is $f(x_1, x_0, y_1, y_0) = \bar{y_1}\bar{y_0} + x_1\bar{y_1} + x_1x_0 + x_1\bar{y_0} + x_0\bar{y_1}$. The corresponding set of cubes for the 2-bit variables (denoted as s_2 because the bit length of each operand n equals 2) is {XX00, 1X0X, 11XX, 1XX0, X10X}. For the cases of larger variables ($n \geq 3$), Code 3.1 directly enumerates cubes from s_{n-1}. It will be proven that any stimuli/cube "a b" in s_{n-1}

Code 3.1 A typical algorithm to generate the set of cubes for the operator '$\geq$'.

```
function GetSetOfGE(integer $n$)                                  ▷ Calculate $s_n$
    if $n = 1$ then
        return $\{1X,X0\}$                                        ▷ Return $s_1$
    end if
    $s_{n-1} \leftarrow$ GetSetOfGE($n-1$)
    $s_n \leftarrow \{1\overbrace{X\dots X}^{n-1}0\overbrace{X\dots X}^{n-1}\}$   ▷ Initialization
    for each cube "$ab$" in $s_{n-1}$ do
        $s_n \leftarrow s_n \cup \{1aXb, Xa0b\}$
    end for
    return $s_n$
end function
```

can be expanded to 2 cubes for s_n by prefixing, i.e., "$1a\ Xb$" and "$Xa\ 0b$".

The algorithm in Code 3.1 generates the cube set s_n for the two n-bit unsigned numbers p and q, such that $p \geq q$. These two numbers can be denoted as $p = \langle p_{n-1}\overbrace{p_{n-2}\dots p_1p_0}^{\text{denoted as } p'}\rangle, q = \langle q_{n-1}\overbrace{q_{n-2}\dots q_1q_0}^{\text{denoted as } q'}\rangle$ in binary format. The initial set is $s_1 = \{1X, X0\}$ (Lines 1-3). The iteration formula is (Lines 6-9):

$$s_{i+1} = \{1\overbrace{X\dots X}^{i\text{'}X\text{'s}}0\overbrace{X\dots X}^{i\text{'}X\text{'s}}\} \cup \{1aXb, Xa0b | ab \in s_i\} \tag{3.5}$$

The proofs for completeness and sufficiency are given next.

First the completeness property is proved by induction.

Proposition: If $p \geq q$, then the pair $\langle pq\rangle$ is implied by a cube from s_n.

Base case: For $n = 1$, all the valid binary pairs (i.e., 11,10 and 00) are implied by the two cubes in s_1 (i.e., 1X implies 11 and 10 respectively, and X0 implies 00).

Inductive hypothesis: Suppose the completeness property holds for up to $n = m$.

Inductive step: For $n = m + 1$, because $p \geq q$, the most significant bits p_m and

q_m should be either $p_m = 1, q_m = 0$ or $p_m = q_m$:

1. If $p_m = 1$ and $q_m = 0$, i.e., $\langle pq \rangle = \langle 1p'0q' \rangle$, then $\langle pq \rangle$ could be implied by the cube $1\overbrace{X \ldots X}^{m\text{`}X\text{'}s}0\overbrace{X \ldots X}^{m\text{`}X\text{'}s} \in s_{m+1}$.

2. If $p_m = q_m$, then $p' \geq q'$ because $p \geq q$. Hence, according to the assumption that the completeness property holds for $n = m$, $\langle p'q' \rangle$ is implied by a cube in s_m. Assuming this cube is ab, then $\langle pq \rangle$ can be implied by the cube $1aXb \in s_{m+1}$ if $p_m = q_m = 1$, or by the cube $Xa0b \in s_{m+1}$ if $p_m = q_m = 0$.

Thereby, the case for $n = m + 1$ holds, which proves the completeness property.

The sufficiency property is also proven by induction.

Proposition: Any pair p, q implied by s_n should satisfy $p \geq q$,

Base case: For $n = 1$, $s1 = \{1X, X0\}$ implies 11, 10 and 00. In each of these cases $p \geq q$ holds.

Inductive hypothesis: Suppose the sufficiency property holds for up to $n = m$.

Inductive step: For $n = m + 1$:

1. The cube $1\overbrace{X \ldots X}^{m\text{`}X\text{'}s}0\overbrace{X \ldots X}^{m\text{`}X\text{'}s}$ implies $p = \langle 1p' \rangle$ and $q = \langle 0q' \rangle$. For each of these pairs, $p \geq 2^m > q$.

2. Any cube in $\{1aXb | ab \in s_m\}$ implies $p = \langle 1p' \rangle$ and $q = \langle q_m q' \rangle$, where $\langle p'q' \rangle$ is implied by $ab \in s_m$. According to the assumption that the sufficiency property holds for $n = m$, we have $p' \geq q'$. Hence $p = 2^m + p' \geq q_m 2^m + q' = q$. The proof for the case of $\{Xa0b | ab \in s_m\}$ follows the same line of reasoning.

Thereby, the case for $n = m + 1$ holds, which proves the sufficiency property.

Table 3.1: The set of cubes for the SystemVerilog constraint from Code 2.3.

`opcode[2:0]`	`opr1[7:0]`	`opr2[7:0]`
`00X`	`XXXXXXXX`	`XXXXXXXX`
`01X`	`XXXXXXXX`	`00000XXX`
`100`	`XXXXXXXX`	`00000XXX`

A similar method can be applied for generating the set of cubes for other types of unary/binary operators. Table 3.1 shows the converted set of cubes from the constraints defined in Code 2.3 using **enum**, **inside** and **if**-implication expressions, in which each cube covers a subspace of the compliant stimuli.

Regarding the issue of programmability, SystemVerilog constraint expressions can be written as class blocks or as in-line constraints and they can be enabled/disabled seamlessly in a testbench in pre-silicon verification environments; in hardware they can also be enabled/disabled via in-system/on-line reprogramming of the CRSG, however it does require user intervention because the new content for the CRSG needs to be regenerated and downloaded into the on-chip memory. Both the content and the size of the equivalent cube set are independent of the hardware architecture and they depend only on the specific user-provided constraint.

The algorithm guarantees the two properties of completeness and sufficiency for the cubes in Section 3.1, when generating the set of cubes from user-defined constraints. Generally, the initial cube set is empty. The algorithm may constantly enumerate a series of compliant stimuli as the pre-silicon verification tool does and add them into the set. The stimuli or original cubes in the set are iteratively merged into new cubes using well-established concepts from two-level logic synthesis, i.e., generation of prime implicants (Coudert, 1994). In this way, the new compliant stimulus could always be added if it is not in or implied by the set. So it guarantees the property of completeness. Since each new cube is derived only by compliant stimuli and cubes, the expansion from the cube can be seen as the inverse process. Hence the property of sufficiency is satisfied. For some cases, where it is impractical to enumerate all the compliant stimuli, the algorithm may directly enumerate cubes proven to be compliant with the type of constraints, as exemplified in Code 3.1.

Because the generated set of cubes from the conversion algorithm may still contain reducible cubes, a two-level logical minimization tool, e.g., Espresso (McGeer *et al.*, 1993), is used to further reduce the cardinality of the cube set.

3.3.2 Converting constraints to cubes using hardware synthesis and BDDs

Taking advantage of the widely available techniques and tools for hardware synthesis and BDD manipulation, as introduced in Section 2.3, the method from this subsection can deal with a variety of types of constraints without the necessity of developing custom algorithms for specific operators.

As emphasized in the previous subsection, the generation of the equivalent set of cubes is similar to the reduction on a K-map of the constraint function. K-maps commonly operate on Boolean functions in the disjunctive normal form (DNF), which is the sum of a products between literals. Each product is mapped to one cube in the equivalent cube set. For instance, the simplified DNF for the K-map in Figure 3.1 is $f = x_1\bar{y_1} + x_1x_0 + \bar{y_1}\bar{y_0} + x_0\bar{y_1} + x_1\bar{y_0}$. The compliant vector space indicated by f is equivalent to the union of the vector set indicated by each cube, i.e., $\{1X0X\} \cup$

$\{11XX\} \cup \{XX00\} \cup \{X10X\} \cup \{1XX0\}$.

The binary decision tree is an equivalent representation of the truth table, which has the same space complexity, i.e., $O(2^n)$, with the truth-table and K-map as introduced in Section 2.3. BDDs reduce the average-case complexity of the decision trees by reordering the variables and merging nodes. A path from the root node to the terminal node of 1 identifies a cube of the function (Somenzi, 1999). For instance, the red path in Figure 2.1(b) implies "010X".

Because the majority of logic constraints in SystemVerilog are functions that can be synthesized to hardware, an alternative approach to cube generation from constraints is to rewrite constraints to a synthesizable Boolean expression. Then, this Boolean expression can be synthesized using any third-party tool and subsequently the cubes can be enumerated from the BDD of the synthesized function.

1. All the enabled user-defined constraints for the current sessions are rewritten into Boolean expressions. Multiple constraints that are simultaneously satisfied are in conjunction with each other. Code 3.2 shows the rewritten results from a single constraint and compounded constraints.

2. Taking advantage of a hardware synthesis engine, the rewritten Boolean function is synthesized into a gate-level netlist. The process includes logic synthesis and optimization (Schliebusch *et al.*, 2010), during which the techniques for converting variables from the constraints to their bit-level equivalent representation is done. For example, Figure 2.2 illustrates the synthesized netlist for the constraint shown in Code 3.2(a).

3. A custom tool can be subsequently built, which can create the BDD representation for the Boolean function that has been synthesized. Any third party

Code 3.2 Rewrite constraints to synthesizable Boolean functions in SystemVerilog.

```
module GreaterEqual_f;
logic [1:0] x, y;
logic f;
assign f=( (x>=y)
         );
endmodule
```

(a) Rewritten from Code 2.1

```
module FPNumber_f;
logic [7:0] exponent;
logic [22:0] fraction;
logic f;
assign f=( (exponent==8'd127) &&
           (fraction[22:21]<=2'd2)
         );
endmodule
```

(b) Rewritten from Code 2.4

BDD package can be leveraged during this step. Cubes can be enumerated by the depth-first-search traversal of the BDD, because a path from the root node to the terminal node of 1 in the BDD identifies a cube of the function (Somenzi, 1999). The work in (Minato, 1996) provides further discussions with more examples on generation of cube sets from BDDs.

In summary, the cube generation method using BDDs eliminates the manual work for developing custom algorithms for different types of constraints, which can be cumbersome especially for compounded constraints. By rewriting constraints into circuit models that are synthesizable, this method can rely on any tools for hardware synthesis and BDD manipulation. Note, since the variable order in BDDs can influence the number of cubes that are generated (and the number of 'X's in the generated cubes), a two-level logic minimizer can be used as a post-processing step to reduce the cardinality of the cube set.

Table 3.2: The dictionary for mapping cube strings into binary cubes.

Character in a cube string	**2-bit mapped binary code**
‘0’	00
‘1’	01
‘X’	10

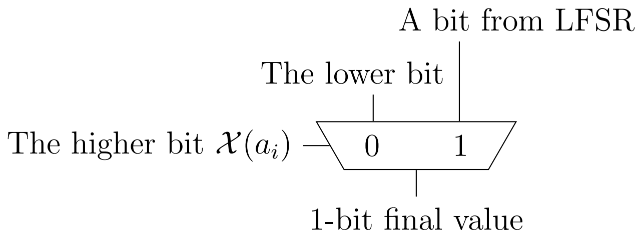


Figure 3.3: Use a 2-1 multiplexer to arbitrate the final bit according to the 2-bit binary code (i.e., the higher bit $\mathcal{X}(a_i)$ and the lower bit).

3.3.3 Hardware-oriented post-processing of cubes

Before transferring them on-chip, the cubes converted from SystemVerilog constraints are mapped into binary cubes based on a simple mapping dictionary shown in Table 3.2. The three valid symbols in a cube, i.e. ‘0’, ‘1’ and ‘X’, occupy three 2-bit binary code points. That is, the higher bit of the code for a given value a_i is $\mathcal{X}(a_i)$, while the lower bit is a_i if $\mathcal{X}(a_i) = 0$, or 0 if $\mathcal{X}(a_i) = 1$. As shown in Figure 3.3, each bit in the final stimulus could be obtained by using a 2-1 multiplexer according to the two bits of the binary code. Hence an m-bit cube is encoded into a $2m$-bit binary word. For instance, the cube “1X0X” is encoded into an 8-bit binary cube 01100010.

A potential shortcoming of using $2m$-bit binary words for each cube is the amount of storage needed for the on-chip memory. Therefore, a compaction process can be used before storing the binary words on-chip. The decoding logic, as elaborated in detail in the contribution from Chapter 4, is used to reproduce the $2m$-bit binary

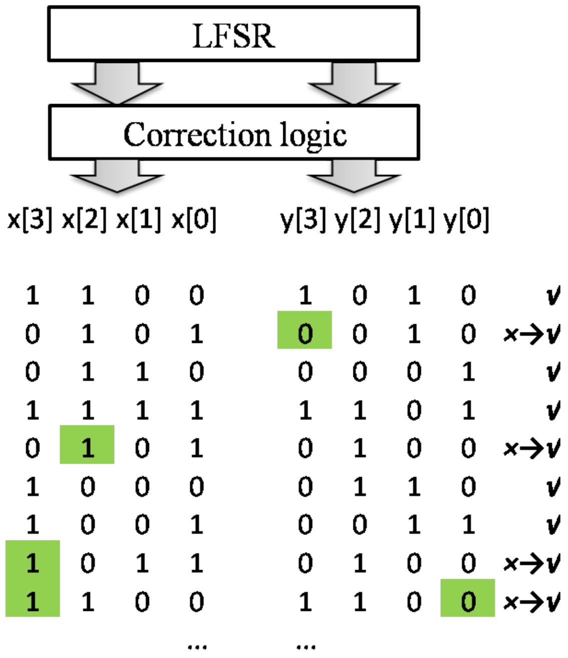


Figure 3.4: Constrained-random stimuli generation by employing correction logic around the LFSR. In order to force the output from the LFSR to satisfy the constraint $x \geq y$, the logic modifies the bits as assigned with non-X value of one cube in Figure 2.3(a).

words on-the-fly, and use them as masks to correct the non-compliant random values from an LFSR, as briefly outlined next.

Compared with the reseeding-based solutions discussed in Figure 2.3, the techniques from this thesis (detailed in the next two chapters) attach correction logic circuitry to the LFSR. Therefore the non-compliant stimuli are not skipped; rather they are corrected. Figure 3.4 exemplifies the correction of the random values from the LFSR, as shown in Figure 2.3(a), into compliant stimuli according to the cubes from Table 2.3 (for the constraint $x \geq y$). Taking the cube "1XXX 0XXX" as an example, it means the output is valid as long as the most significant bits of x and y are 1 and 0 respectively. For example, the output "0011 0100" from Figure 2.3(a) is corrected to “1011 0100” in Figure 3.4 based on cube “1XXX 0XXX”.

The constrained-random stimuli generator (CRSG) elaborated in the following chapters, conceptually uses cubes to *mask* the invalid stimuli at the output of the LFSR, as illustrated in Figure 3.4. This CRSG is in-system programmable, and the user can apply different constrained-random sequences based on distinct user-defined constraints, by updating the on-chip memory (that stores the cube information in compact format) with the content generated from the revised constraints.

3.4 Summary

The fundamental principles of the cube-based representation for constraints have been discussed in this chapter. Two different approaches for translating constraints from a pre-silicon verification environment into cubes have been presented. Using cubes as on-chip masks for the random sequences from an autonomous LFSR, is the basic concept that enables the constrained-random stimuli generators discussed in this thesis. Dealing with the ramifications of using this basic concept (i.e., large amount of on-chip storage or repeating the same stimuli during generation) are the topics that are investigated in the following two chapters.

Chapter 4

Stimuli generation for functional constraints using compact binary cubes

SystemVerilog allows users to specify constraints for constraint-driven stimuli generation. Using these constraints, simulators automatically generate constrained-random stimuli for functional verification. The previous chapter has introduced the cube-based representation for constraints, which can be adapted to a hardware implementation. In this chapter, the technical details of the new constrained-random stimuli generator (CRSG) are introduced. The constrained-random stimuli are expanded on-the-fly and the constraints can be revised by reprogramming the CRSG in-system. To reduced the amount of data to be stored on-chip, a decoder can be integrated into the CRSG. The proposed solutions support both logic constraints (Section 4.2) and sequential constraints (Section 4.3), both of which are compliant to the SystemVerilog standard.

4.1 The overview of the on-chip stimuli generator

The proposed solution operates at both design time and run time (or validation time), as illustrated in Figure 1.2. At design time, the configuration of the CRSG hardware is selected, including, for example, whether it requires support for logic constraints or mixed-type (both logic and sequential) constraints, the capacity of the on-chip memory as well as the dimension and characteristic polynomial of the LFSR. This step is illustrated by the "Insert CRSG" box in Figure 1.2. At validation time, the user is given the freedom to change the configuration of the CRSG, in order to apply functionally-compliant sequences with different (user-programmable) constraints. These constraints for the stimuli are captured in SystemVerilog (i.e., the same language used during the pre-silicon verification) and can be updated iteratively based on the specific debugging needs as the validation process evolves. The constraints are converted into binary cubes or compact binary cubes (CBCs) as illustrated by the "Prepare cubes for validation" box in Figure 1.2. Then the cubes are loaded into the on-chip memory and activated for stimuli correction thus facilitating continuous functionally-compliant random stimuli generation, which is illustrated in box "Start/Update stimuli generation" box in Figure 1.2.

The top-level architecture for the on-chip hardware is sketched in Figure 4.1. It includes four main parts: on-chip cube memory, decoding logic if compact-binary cubes (CBCs) are used, the pseudo-random generator based on a maximum-length LFSR, and a correction structure comprising multiplexers. The control logic manages the addressing strategy for the memory and status of the other logic blocks. The CBCs are stored into the cube memory preceding on-chip stimuli generation. The addressing for the cube memory is handled by the control logic. The pseudo-random

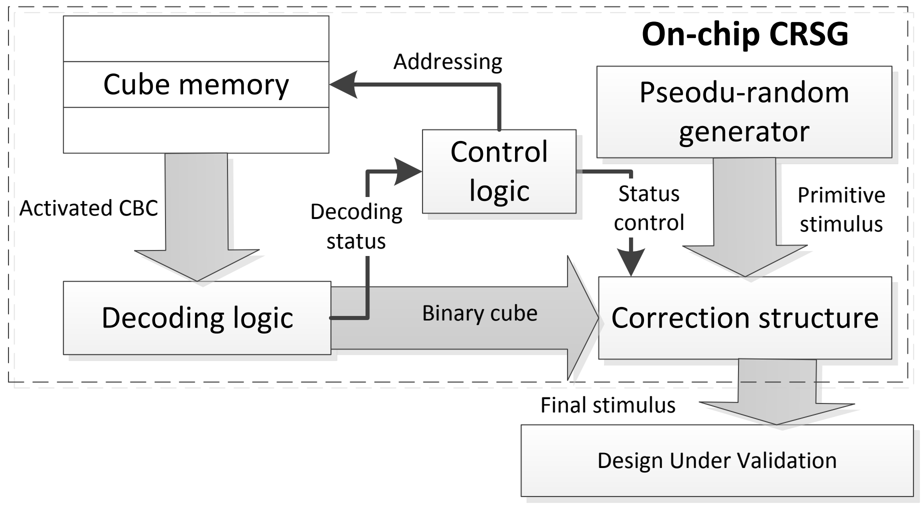


Figure 4.1: The general top-level architecture for the on-chip CRSG.

generator can generate one primitive stimulus per clock cycle. Meanwhile, one CBC is activated each time, which is first fetched from the cube memory and then decoded into a binary cube. Based on the activated cube, the correction logic simultaneously checks the primitive stimulus from the pseudo-random generator and modifies it into a compliant stimulus. Hence, the CRSG can continuously generate compliant stimuli to the design-under-validation (DUV) on a cycle-by-cycle basis.

This chapter provides the details and the variations of CRSG hardware according to the types of constraints that are supported. The solution in Section 4.2 is designed specifically for logic constraints, and the one in Section 4.3 supports sequential constraints. It is worth mentioning that the solution for sequential constraints also supports logic constraints since they are treated as the sequential constraints over a single clock cycle.

4.2 The solution for logic constraints

The logic constraints define invariant formats, values and other features of the stimuli. This section elaborates the solution for logic constraints, including both generating compact binary cubes in the content processing phase, as well as the on-chip hardware circuitry needed to handle these cubes to generate functionally-compliant sequences.

4.2.1 Content processing of cubes with compaction

The user-defined logic constraints in SystemVerilog are converted into binary cubes, according to the flow explained in Section 3.3. That is, the custom software tools translate user-defined constraints into the equivalent set of cubes, encode them to binary cubes, and partition them for compaction. In the following, the compaction format of CRSG for logic constraints is elaborated, which balances the compaction efficiency and the hardware cost for the decoding logic.

As an optional process, the binary codes can be compacted into CBCs. The motivation is that the cube length is shortened and the size of data required for transmitting the cube data to the DUV is reduced. The opportunity of compaction is revealed through investigating consecutive sequences in a cube.

Some cubes may include consecutive-‘X’s, consecutive-‘0’s or consecutive-‘1’s sequences. The cause can be explained by practical requirements of verification/validation, e.g., if a variable is not constrained, all the bits for this variable are filled with consecutive ‘X’s in the cube. Likewise, the consecutive-‘0’ can be used for resetting a variable under some user-defined conditions. Based on this observation, the general compaction algorithm is developed for encoding the binary cubes into CBCs. It adapts the run-length coding strategy, by accounting also for the following issues:

- In-system programmability: A key feature of the solution proposed in this chapter is the ability to change the constraints during post-silicon validation and reprogram the CRSG in-system. Hence, the encoding method should not depend on a particular cube set, rather it should be capable of supporting any set of cubes that can be derived from any constraints that are specified by the user after the fabrication of silicon prototypes.

- Low area cost for the on-chip decoding logic: Because CBCs are decoded by on-chip hardware, the decoding process should be simple so that the hardware area is not excessive. Using a run-length decoding approach is intuitive and it does not require extra on-chip random access memory (RAM) for storing auxiliary information, e.g., dictionaries for decoding.

- Word alignment: Considering that CBCs are loaded to on-chip RAMs, the fixed-length segments in each CBC that are consistent with the word length of RAMs could help eliminate the extra hardware for dealing with in-word offset. That is, each CBC could be fetched from the RAM word by word no matter what the content of the word is. As shown in the following, run-length codes used in this chapter could be aligned to 8/16-bit word.

- High-throughput: In order to achieve high throughput of stimuli generation, the decoding process should be configurable to decode an entire CBC in a short time, e.g., as fast as one cube per cycle. Thus those compaction methods that include multiple steps, e.g., sliding windows, are not applicable. Since the words encoded using the run-length process are independent, they can be decoded in parallel in a single cycle.

The algorithm for encoding binary cubes uses a run-length encoding approach, which is suitable for hardware decoding with low area and high-throughput. It first partitions a cube into segments, either run-length segments or mixed segments. The consecutive sequence longer than a threshold is partitioned as a run-length segment, comprised of a r-bit binary number denoting the segment length and a 2-bit binary code prefix denoting the consecutive character according to Table 3.2. For instance, given $r = 6$, a consecutive sequence "11111" in a cube can be compacted as $\overbrace{01}^{\text{prefix}}\,\overbrace{000101}^{\text{length}}$, which is shorter than the original encoded binary 0101010101. Otherwise the sequence between two run-length segments is partitioned as a mixed segment. The threshold depends on the configuration of the on-chip hardware, so as to make a consecutive sequence compacted shorter by being partitioned as a run-length segment than as a mixed segment.

Regarding the value of r, it is mainly configured based on the average length of consecutive codes (i.e., consecutive-'X's, consecutive-'0's and consecutive-'1's) in cubes, in addition to the consideration of word alignment for the RAM. For instance, setting $r = 6$ is suitable for most cases in which the length of stimuli is approximately in the range from 20 to 100, because the average length of consecutive codes is most likely less than 64 bits for these cases according to the constraints. A very large r may lead to more wasted bits in the run-length segment for most cubes; on the other hand, if r is too small, it may result in more run-length segments which could have been compacted in one segment with less data size.

Provided the binary field for the segment length is r bits, a binary cube can be compacted into a CBC down to the size of $1/2^{r+1}$ of the original binary cube. On the other hand, the worst case is when the binary cube is a single mixed segment.

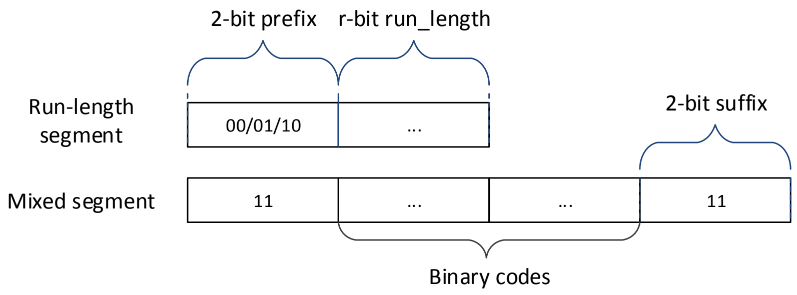


Figure 4.2: The format for the run-length segment and the mixed segment for cube compaction.

Table 4.1: An example of CBC based on the logic constraints shown in Code 2.3.

Cube	**Binary cube**	**CBC** ($r = 6$)

`00X`
`XXXXXXXX`
`XXXXXXXX`
	000010
	1010101010101010
	1010101010101010
		11000011
		10010001

`01X`
`XXXXXXXX`
`00000XXX`
	000110
	1010101010101010
	0000000000101010
		11000111
		10001001
		00000101 10000011

`100`
`XXXXXXXX`
`00000XXX`
	000110
	1010101010101010
	0000000000101010
		1101000011
		10001000
		00000101 10000011

Figure 4.2 shows the compaction format for the two types of segments. The run-length segment for the consecutive sequence includes a 2-bit prefix denoting the consecutive character longer than a threshold, and an r-bit binary value denoting the run length of the sequence. The mixed segment denoting the non-consecutive sequence is filled with the original binary codes and edged with a 2-bit prefix and a 2-bit suffix equal to 11. The compaction results for the cubes in Code 2.3 are shown in Table 4.1, in which case the threshold is 2 and $r = 6$.

4.2.2 On-chip CRSG architecture

The proposed CRSG consists of the control unit, the on-chip cube memory, the decoding logic, the pseudo-random generator and the correction structure, as shown in Figure 4.3. It supports the workflow from storing and decoding CBCs, primitive random stimuli generation and correction, to producing the final stimuli output. Both the throughput and memory logic can be flexibly adapted to the validation environment. The unconstrained stimulus from the pseudo-random generator is biased by the correction structure that gets the activated binary cube from the cube memory and the decoding logic.

Cube memory

The on-chip cube memory stores the CBCs. When the control unit activates a CBC, its initial address is sent to the cube memory. The fetched CBC is loaded into the reading buffer for decoding. Considering the issue of word alignment for the memory, each CBC starts with a new address, so that no in-word offset information is needed for activating a new CBC.

The CRSG only requires a small on-chip cube memory for buffering a few CBCs, provided the subsequent CBCs can be uploaded via a low-bandwidth interface from a host. Supposing that the stimuli were transmitted directly from a host or an on-board memory, i.e., without compaction and on-chip buffering, then each stimulus would be used once and discarded. Such mechanism would require new stimuli frequently and its main limitation is the need for very high-bandwidth interfaces. By contrast, by employing an on-chip embedded memory for buffering CBCs, the proposed CRSG architecture receives compact cubes, which are then expanded on-the-fly to correct

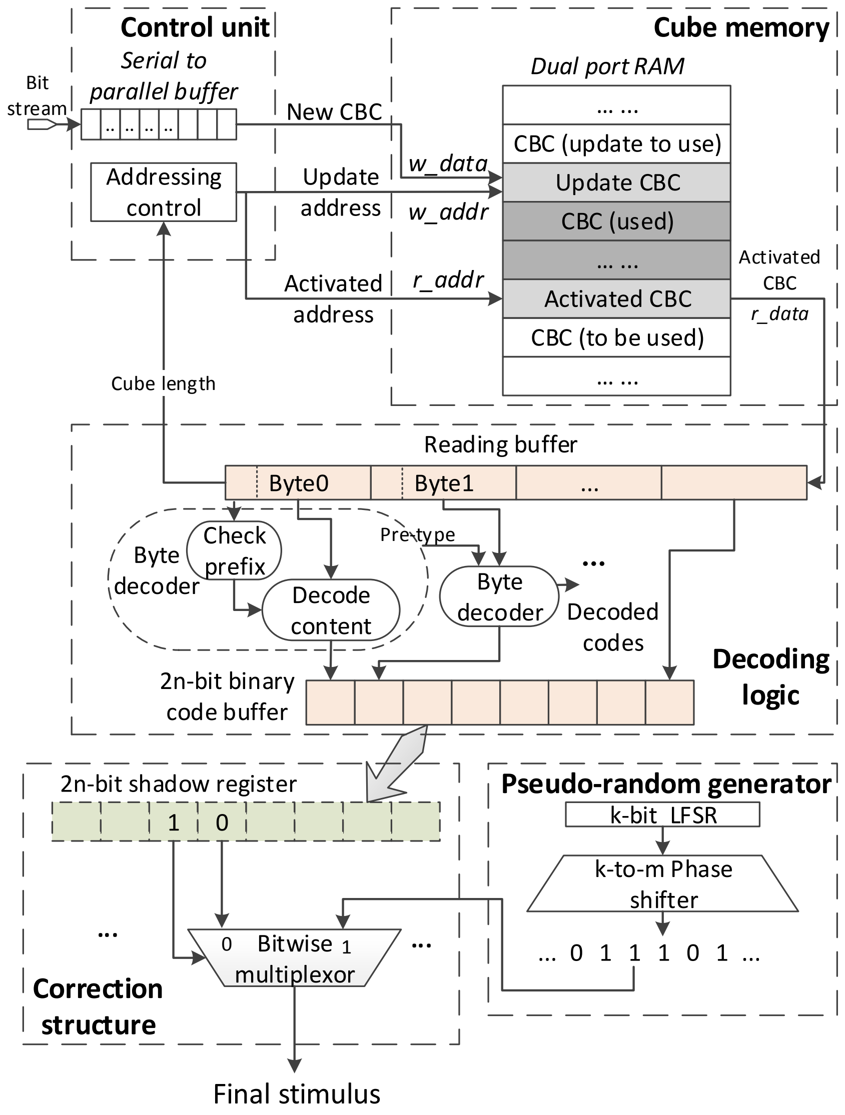


Figure 4.3: The architecture of the on-chip CRSG for logic constraints.

LFSR patterns during the stimuli application. Each CBC can remain activated for an arbitrary number of cycles to constrain the pseudo-random generator to generate a user-controlled amount of valid stimuli. Because both the number and the size of CBCs are much smaller than the expanded stimuli, it alleviates the need for high-bandwidth interfaces.

The cube memory can be implemented either as a first-in, first-out (FIFO), i.e., addressed by implicit increment, or a dual-port RAM. In the case the volume of CBCs is very large, the capacity of the embedded memory can be lowered by buffering only a subset of CBCs, which will be used during a limited time window for stimuli application; the subsequent subset of CBCs can be uploaded concurrently with the application of the stimuli expanded from the current subset of CBCs. Finally, it should be noted that the capacity of the cube memory is not influenced by the circuit size, since it is determined only by the number and dimension of cubes, which are influenced by the type of constraints and the size of randomized packets.

Figure 4.3 shows the data flow for activating and updating CBCs based on the cube memory built with dual-port RAM. One port is used for fetching the activated CBC via *r_addr* and *r_data*. The other port including *w_data* and *w_addr* is reserved for updating a new CBC when it is ready from the control unit. The bitstream transmission via a low-bandwidth interface takes less pin resources, while the control unit reconstructs the CBC and sends word by word to the cube memory. The recently sent CBC is updated and after being activated and used, the CBC can be set to be outdated and can be overwritten by a new CBC. The addressing control unit issues the CBC address for update and activation independently. The control unit keeps track of the activated address and the update address where the used CBC can be overwritten by a new CBC transmitted from the host.

Decoding logic

The decoding logic consists of multiple byte-wise decoders to support parallel decoding and a $2n$-bit buffer to store the decoded n-code binary cube, as shown in Figure 4.3. It supports to decode p binary codes (each binary code has 2 bits) per clock cycle, where p denotes the degree of decoding parallelism. Each combinational byte-wise decoder determines the segment type by the 2-bit prefix (if the type is not inherited from the previous byte). The byte is interpreted into 2 to 4 codes as a mixed segment or 2 to 2^r codes as a run-length segment.

In each clock cycle, the first p codes from parallel decoders are shifted into the binary codes buffer, leaving the remainders for the following cycles. Thus decoding a CBC of n codes requires $\lceil n/p \rceil$ cycles. For example, a 168-code binary cube (as used in the experiments detailed in Section 4.4) is decoded from the CBC format in 21 cycles if p is 8, or 11 cycles if p is 16. The parallelism facilitates rapid continuous cube switching.

Pseudo-random generator and correction structure

The pseudo-random generator consists of a k-bit maximum-length LFSR and a k-to-m phase shifter ($k \leq m \leq n$), as shown in Figure 4.3. The phase shifter consists of combinational logic with `XOR` gates, which expands each k-bit output from the LFSR to m-bit primitive stimulus.

The correction structure consists of a $2n$-bit shadow register and m bitwise multiplexers, as shown in Figure 4.3. Each two bits in the shadow register are paired with a bitwise multiplexer. The shadow register pipelines n decoded binary codes from the decoding logic, which avoids stalling stimulus correction during cube switching. A virtual $2m$-bit window is created and rolled in the $2n$-bit shadow register, which indicates the m activated binary codes for the current cycle. Each multiplexer decodes a 2-bit binary code in the virtual window and arbitrates whether to output the corresponding bit from the pseudo-random generator or to correct it to a constant 0 or 1. Based on the encoding dictionary, the higher bit in the binary code can directly serve as the selection signal and the lower bit is the constant output when it is corrected.

Control unit

The control unit keeps track of addresses for the cube memory, as shown in Figure 4.3. It uses the cube memory as a circular queue. Both the activated address and the update address move down to the following CBC position until the end of the memory, or they reset to the initial address. In order to compute the address for the next activated CBC, the control unit receives the length of the currently activated CBC from the decoding logic, which is added to the current activated address. The activated CBC is fetched from the cube memory and decoded into a binary cube by the byte decoders. Then it is copied to the shadow register in one cycle, based on which the multiplexers arbitrate each output bit between the pseudo-random bit and the lower bit in the mask code.

The control unit also synchronizes the functional parts, so that the architecture supports to generate an n-bit final stimulus (or packet) in a user-specified number of clock cycles (denoted as T). Therefore the stimulus is split into m-bit slices (m is equal to $\lceil n/T \rceil$), except the last slice if the remainder is not zero. As shown in

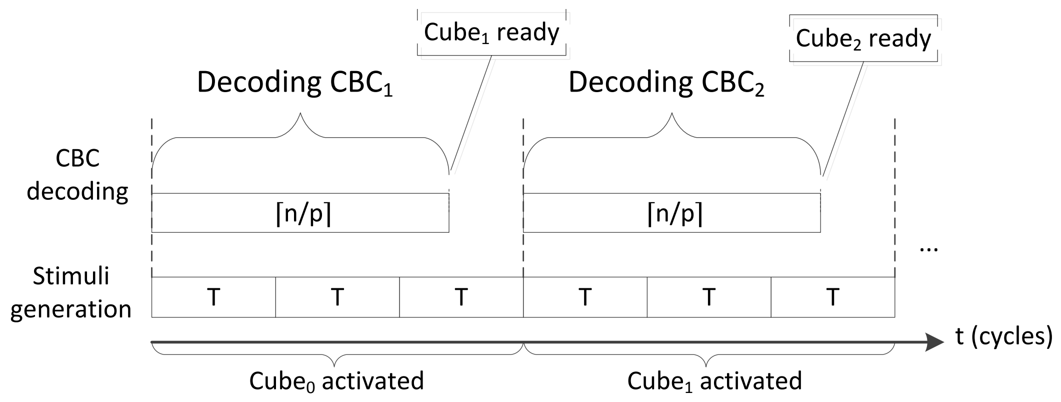


Figure 4.4: Timeline for CBCs decoding that influences how frequently a CBC can be switched.

Figure 4.4, three packets are generated within $3T$ cycles based on Cube$_0$. Meanwhile CBC$_1$ is decoded into Cube$_1$ and will be activated after the third packet is completely generated. Generally, while an n-bit packet is generated in T cycles, the next CBC is being decoded and will be ready within $\lceil n/p \rceil$ cycles. Then it switches to be active immediately after the previous complete packet is generated. Figure 4.4 illustrates the minimum cycle requirement for switching to a new cube, within which $\lceil n/pT \rceil$ packets must be generated based on the same cube.

4.2.3 The distribution of stimuli based on the architecture

The distribution of the stimuli generated by the CRSG relies in part on the uniform distribution associated with LFSR based on primitive characteristic polynomials, however it also strongly dependent on the position of ‘0’s, ‘1’s and ‘X’s in each cube; in addition, the distribution is also biased by the state of the LFSR when a particular CBC is activated. Taking a 4-bit LFSR as an example, if the LFSR has two adjacent states “1100” and “1001”, i.e., the LFSR shifts left by one position and it feeds ‘1’ at the rightmost position, an unconstrained CRSG (i.e., the activated cube is “XXXX”) will output the two binary values as two consecutive stimuli; hence the distribution of samples at the output of the LFSR will contain each of the samples “1100” and “1001” exactly once. If the activated cube is “XX10”, the two generated stimuli are “1110” and “1010”, thus each of them will again count once in the distribution. However, if the activated cube is “X1X0”, both stimuli from the output of the LFSR will be corrected to “1100” and hence this particular sample will be accounted for twice in the distribution. The impact of the correction logic on the sample distribution is experimentally assessed in Section 4.4.

4.3 The solution for sequential constraints

The solution in the previous section considers only logic constraints. In many applications, nonetheless, it is important to use sequential constraints to specify the behaviour in adjacent cycles in the stimuli, as outlined in Chapter 2. The solution in this section expands the solution for logic constraints to support sequential constraints. The distinguishing features including the cubes with timing information and the new generator are emphasized.

4.3.1 Cubes with timing information

As elaborated in Section 2.4, the sequential constraints can be seen as a series of partial logic constraints in a specific order, which are activated one by one in adjacent cycles during stimuli generation. Therefore, the general conversion flow for cubes in Section 3.3 is reused for each partial logic constraint, and the generated partial cubes are combined in the order specified in the sequential constraint.

For logic constraints, each cube in the set has the same length as the final (constrained) stimulus. For sequential constraints, an m-bit stimulus (either a packet or a slice) must satisfy constraints in each clock cycle for T consecutive cycles. Hence, the accumulated length of the stimuli in a complete test case is $m \times T$. Therefore, each cube in the equivalent set for such scenarios is $n = m \times T$ bits. Using only logic constraints can be considered as the case with $T = 1$.

4.3.2 Loose-coupling compaction for cubes

Similar to processing the cubes for logic constraints, each '0', '1' and 'X' in a cube are encoded with three 2-bit binary codes: 00, 01, and 10 respectively. Regarding the compaction, the algorithm for logic constraints generates CBCs in order to reduce the required on-chip memory capacity. Nevertheless the side effect of that algorithm introduced long path delays because the decoding logic for the compacted cubes has to check the prior byte before decoding the current byte, as shown in Figure 4.3. Hence, a new loose-coupling compaction algorithm is proposed for sequential constraints, where all bytes within the cube can be decoded independently. It consists of serial compaction by customized software and on-chip parallel restoration.

Each cube is partitioned into two types of segments, i.e., the run-length segment and the mixed segment. Different from the compaction format varied with the run length r for logic constraints shown in Figure 4.2, the mixed segment is redesigned to contain a fixed number of characters (except for the last segment in the cube). It has a 2-bit prefix 11 followed by the binary codes. The value of r in the run-length segment and the number of characters in the mixed segment are adapted to the word

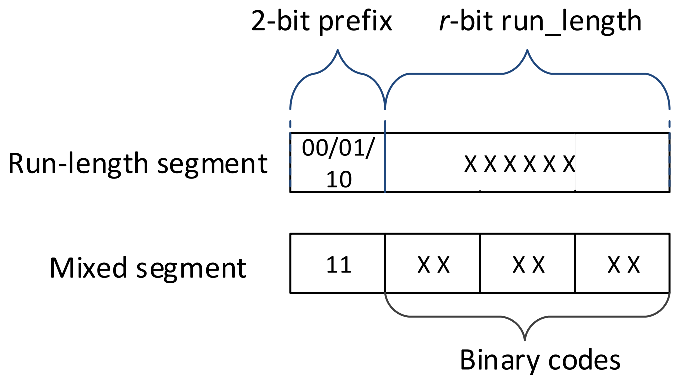


Figure 4.5: The two types of segments ($r = 6$) for sequential constraints.

alignment strategy of the on-chip memory. Figure 4.5 shows an example for the byte-wise segment format (i.e., an 8-bit word), where r equals 6 and each mixed segment contains 3 binary codes.

For the byte-wise segment format, each byte in the CBC format expresses either a run-length segment or a mixed segment independent of other bytes. Table 4.2 shows the resulted CBCs from the constraints in Code 2.3, which are different from the results in Table 4.1. The set of CBCs are loaded in the on-chip memory. Since the set of cubes is equivalent to the original SystemVerilog constraints, the CRSG is therefore programmed to continuously generate compliant stimuli. The details of the generator are described next.

4.3.3 On-chip CRSG architecture

As shown in Figure 4.6, the on-chip generator follows the hardware frame depicted in Figure 4.1, which contains four parts, i.e., the on-chip memory with addressing control logic, the decoding logic, the pseudo-random generator and the correction logic. Since the previous generator for logic constraints shown in Figure 4.3 cannot support partial cubes, the distinguishing features of this new generator stem from

Table 4.2: The CBC using loose-coupling compaction.

Cubes	Binary cubes	CBCs
00X XXXXXXXX XXXXXXXX	000010 1010101010101010 1010101010101010	11000010 10010000

01X
XXXXXXXX
00000XXX
	000110
	1010101010101010
	0000000000101010
		11000110
		10001000
		00000101 10000011

100
XXXXXXXX
00000XXX
	010000
	1010101010101010
	0000000000101010
		11010000
		10001000
		00000101 10000011

using a single clock cycle for decoding and correction. Because the new architecture supports decoding of a cube within one cycle, there is no need to use buffers to store the intermediate parts of a decoded cube; rather, all the parts of a single cube are assembled together within the same clock cycle using the combiners tree highlighted in Figure 4.6. In particular, the decoding logic and correction parts are redesigned as pure combinational logic, which eliminate the cycle delays from fetching a cube to generating stimuli based on it. In each cycle, a CBC is fetched from the memory and decoded, which is then used to correct the LFSR values using the correction logic.

Cube memory and addressing control logic

The on-chip memory is used to store CBCs, which is similar to the memory shown in Figure 4.3. The cubes can be updated in-system, so as to fulfill the programmability feature of the CRSG. The address control unit issues the address for the current cube (i.e., the activated cube) and computes the following address based on different scenarios. In each cycle, only one cube is activated for decoding, and the stimulus generated in this cycle is based on the activated cube. The CRSG supports switching

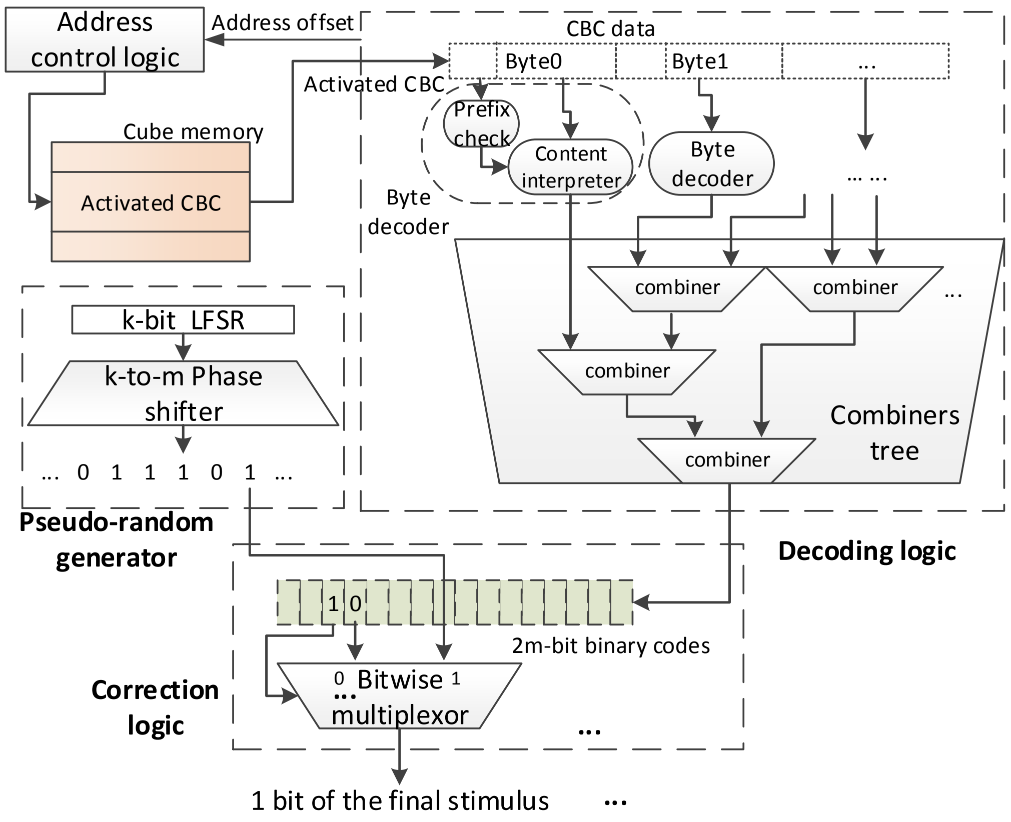


Figure 4.6: The architecture of the generator for sequential constraints.

the activated cube as fast as in a single clock cycle.

Figure 4.7 illustrates the decoding timeline for the sequential constraints scenario. It generates a complete n-bit stimulus within T cycles, during which the partial cubes are switched in order and an m-bit ($n = mT$) stimulus is generated. After the generation of a complete n-bit packet, this control unit can issue the address to the next cube or repeat the previous one.

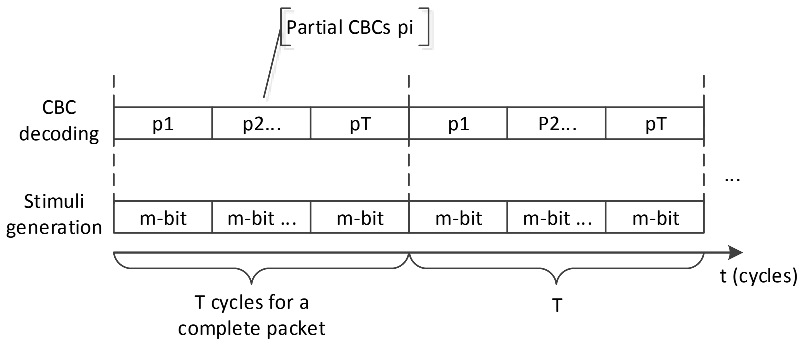


Figure 4.7: The timeline for generating stimuli with partial CBCs.

Decoding logic with combiners tree

The decoding logic is used to restore the cubes in the compact format to the binary cube format. Compared with the decoding logic for logic constraints shown in Figure 4.3, this decoding logic has been redesigned in order to fulfill the requirements for sequential constraints, including fast switching between partial cubes used for the correction of LFSR sequences. The decoding logic consists of a series of byte decoders and a combiners tree, both of which are combinational. Hence it can decode the CBC (or a part of it when using sequential constraints) in the same cycle when the cube is fetched from the memory.

Each byte decoder receives one byte from the activated CBC and decodes it into binary codes. It first checks the 2-bit prefix for the segment type, which decides the interpretation of the following bits. The number of decoded binary codes depends on the length of the word. In the current implementation (the length of the run-length field r shown in Figure 4.5 equals 6), each byte decoder generates 3 binary codes for a mixed segment or 4 to 64 binary codes for a run-length segment (note, the last byte can be decoded into less than 3 valid binary codes). The number of byte decoders depends on the longest CBC. In theory, any binary cube for m-bit stimuli can be compacted into a CBC with the length less or equal to $\lceil 2m/r \rceil$ words. Each word contains $r+2$ bits, or $\lceil (r+2)/8 \rceil$ bytes for the byte alignment. For the case of a byte decoder in which $r = 6$, the longest CBC is not longer than $\lceil m/3 \rceil$ bytes.

Each byte decoder is specialized according to its position in the decoding logic. Given that each partial cube contains at most m binary codes ($m \leq 64$), the maximum number of output binary codes from each byte decoder can be decreased by 3 from the first to the last one, because each preceding decoder offers at least 3 binary codes. Although it might also work by using homogeneous byte decoders with the same number of the output binary codes, the heterogeneous implementation would consume less on-chip area. Since the area of the byte decoder is proportional to the number of the output binary codes (suppose the unit area of each byte decoder equals Cm, in which C is a constant and m is the number of the output binary codes of each byte decoder), the area cost of all the byte decoders can be decreased from $Cm \times \lceil m/3 \rceil \approx Cm^2/3$ to nearly half of it, i.e., $Cm+C(m-3)+C(m-6)+\ldots+C(m \bmod 3) \approx Cm^2/6$.

The combiners tree shown in Figure 4.6 collects the outputs from all the byte decoders and combines them into m binary codes. The tree is created as a balanced binary tree to minimize the path delay. The leaf nodes are the byte decoders, and the inner nodes are combiners. Each combiner combines the outputs from its two children into one. In addition, the combiner tree counts the length of the current CBC, which is sent to the address control unit for computing the next address.

As it was shown in this section, and as it will be demonstrated by the experimental results from Section 4.4, the compaction strategies based on segmentation influence the area cost for the on-chip decoding hardware and the speed of decoding, as it is required by the types of constraints that are supported. That is, the compaction strategy of the CRSG for logic constraints presented in Section 4.2 requires less area for decoding, however it needs multiple cycles to process a CBC. The CRSG for sequential constraints from this section is based on a loose-coupling compaction strategy with fewer cycles but more area cost.

Pseudo-random generator and correction logic

The pseudo-random generator comprises a k-bit maximum-length LFSR and an optional k-to-m phase shifter. The correction structure consists of m independent 2-to-1 multiplexers, and each of them is connected to one bit from the m-bit LFSR value and 2 bits from the m 2-bit binary codes from the decoding logic. Compared with the correction structure from Figure 4.3, the new structure eliminates the shadow register.

4.4 Experimental results

In this section, the cost of the proposed CRSGs and their effectiveness are examined in applying extensively long random, yet functionally-compliant, sequences during post-silicon validation. The results are compared with the methods published in the public domain, which work on the same topic of constrained-random stimuli generation for post-silicon validation using programmable on-chip signal generators that can be applied to any logic block. The proposed generator for logic constraints in Section 4.2 is compared against the method in (Kinsman *et al.*, 2013), which tackles exactly the same challenge of designing and applying user-programmable constrained-random sequences in real-time. The performance of the generator for sequential constraints in Section 4.3 is also examined in this section.

4.4.1 The hardware evaluation of the generator for logic constraints

The proposed CRSGs for logic constraints and sequential constraints are analyzed by varying the length of the LFSR (denoted as k-bit) and the length of the final stimulus (denote as n-bit). Note, the length of the stimuli should support the maximum size of random packets for the different blocks that are validated using the respective CRSG. For this experiment, it is assumed that all the stimuli bits for the entire packet are applied in a single clock cycle. The hardware cost is assessed assuming the CRSG is deployed adjacent to the design-under-validation (DUV), in which case the area cost for the interconnection between CRSG and DUV is negligible and thus not taken into account in the results.

The synthesis results based on the k-bit LFSR are shown in Figure 4.8, in which the results for both CRSG architectures are illustrated. Compared with the reference design (Kinsman *et al.*, 2013), whose area is directly influenced by the size of the LFSR, the area of the proposed CRSG architectures grows insignificantly with the length of the LFSR. Considering that the period of the LFSR-generated sequence has an exponential dependence on the dimension of the LFSR (assuming the characteristic polynomial of the LFSR is primitive and irredundant), the proposed CRSG architecture can employ large LFSRs to avoid the repetition of pseudo-random stimuli when very long validation times are needed; for example, even a 50-bit LFSR that works at 1 GHz can operate autonomously for over ten days.

The synthesis results of the architecture according to different lengths of stimuli

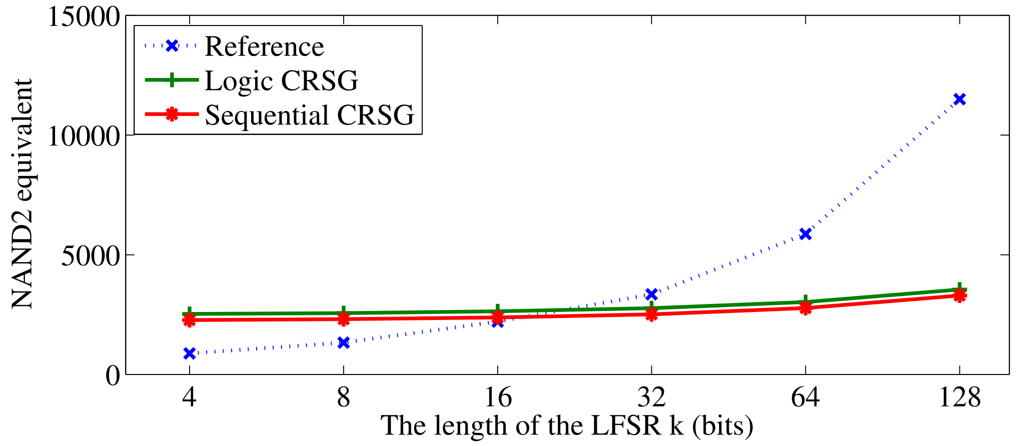


Figure 4.8: The hardware cost of proposed cube-based CRSGs and the reference design in (Kinsman *et al.*, 2013) according to the length of LFSR k. The proposed CRSG for logic constraints is set with $T = 1$, $n = 16$ and $p = 8$, and the one for sequential constraints is set with $T = 1$ and $m = 16$.

are illustrated in Figure 4.9. Due to the $2n$-bit binary code buffer and the $2n$-bit shadow register (see Figure 4.3) and unlike (Kinsman *et al.*, 2013), the proposed CRSG is dependent on the size of the validation stimuli (packets) that are applied to the design-under-validation. Note, however it is common that large packets are broken into smaller ones over multiple clock cycles, which is handled by the generator for sequential constraints (as discussed in sub-section 4.4.2).

As discussed in Section 4.2, an important parameter, which influences how fast the cubes used for correction of pseudo-random sequences can be switched, is the degree of decoding parallelism p. The synthesis results by varying p are illustrated in Figure 4.10 and Figure 4.11. As for the previous experiments, it is assumed that the stimuli are applied in a single clock cycle. As expected, both the area and the critical path delay are affected by p, because each byte decoder must decide the segment type based on the previously decoded byte or the current byte prefix (as described in Section 4.2). The parameter p is independent of the complexity of

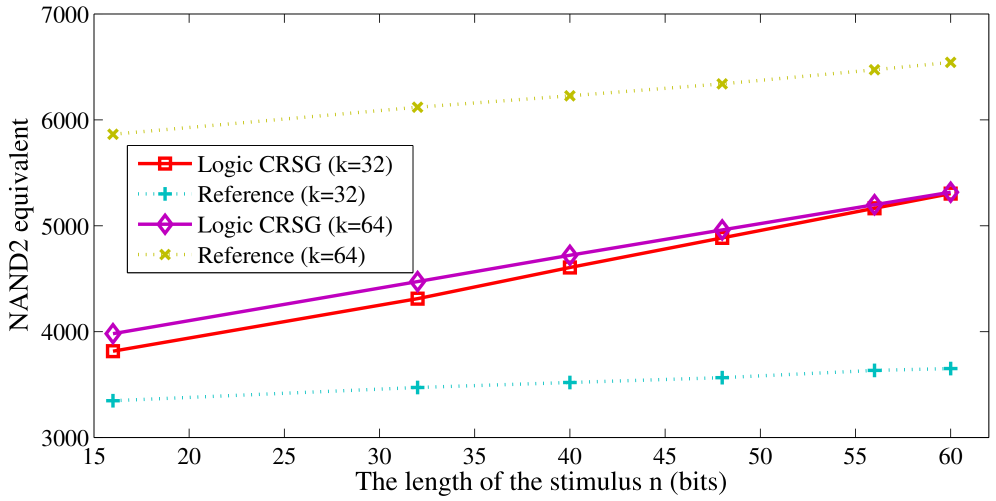


Figure 4.9: The hardware cost of the proposed CRSG for logic constraints and the reference design in (Kinsman *et al.*, 2013) according to the length of the stimulus n (given $p = 8$).

specified constraints, which might influence only the number of mask cubes that are compacted and stored in the on-chip memory. Nonetheless, the higher the degree of parallelism, the faster one can switch between these cubes.

Concerning the side-effects of the proposed CRSG architecture on timing, when one considers the whole view of on-chip functional units and interconnection logic, the delay paths in the CRSG are unlikely to dominate the circuit's operating frequency. What the CRSG architecture impacts is the timing delay from original function signal to the port of the design, which is now multiplexed between the original signal and the stimulus from CRSG. In the event that CRSG will impact the operating frequency, an optional n-bit pipeline register chain can be inserted between the output of the correction structure and the target design.

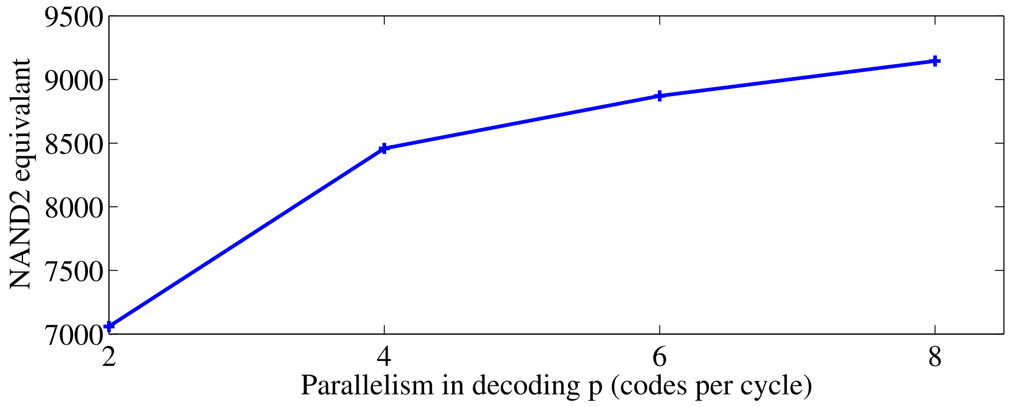


Figure 4.10: The hardware cost of the proposed CRSG for logic constraints according to the degree of parallelism in the decoding logic (given $T = 1$, $k = 168$ and $n = 168$).

4.4.2 The hardware evaluation of the generator for sequential constraints

For the CRSG for sequential constraints, the number of byte decoders is varied according to the maximum length of the CBC. This number is determined by the length of stimulus per cycle (denoted as m-bit), as well as the compaction efficiency. Figure 4.12 illustrates the relation between the area and m. Furthermore one can argue that for most protocol-based interfaces data is transferred over multiple clock cycles; thus any large packet can be broken into slices with a low value of m (using sequential constraints over consecutive slices) and therefore can be supported by the proposed method with relatively low hardware overhead.

Regarding the maximum frequency of the proposed CRSG, the memory access latency is considered constant and the path from the LFSR to the corrected stimulus contains the `XOR` network and one 2-to-1 multiplexer. Hence, the critical path delay in the proposed CRSG is mainly dependent on the number of byte decoders, which

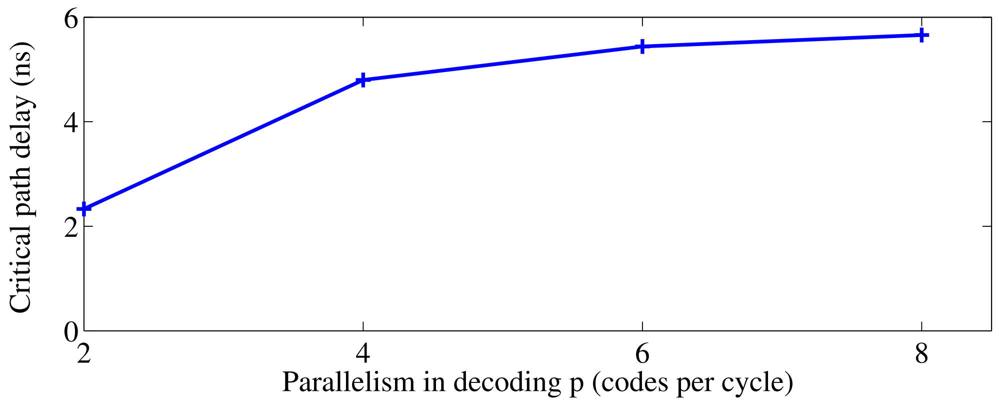


Figure 4.11: The critical path delay of the proposed CRSG for logic constraints according to the degree of parallelism in the decoding logic, estimated by static timing analysis with a CMOS 90nm standard cell library (given $T = 1$, $k = 168$ and $n = 168$).

impacts the depth of the combiners tree from Figure 4.6. In Figure 4.13 the relationship between the number of the decoders adapted to m and the critical path delay is illustrated. When m grows, the number of byte decoders (needed to decode CBCs within one clock cycle) increases; thus the depth of the combiners tree also grows causing an increase in the critical path delay.

4.4.3 The evaluation on the data volume of the cube set

Concerning the impact of constraint complexity on the cube memory, a series of experiments are performed using integer linear programming (ILP)-type constraints because they can intuitively illustrate the numerical relationships between variables (and many arithmetic, relational and even some logic constraints can be converted to ILP forms). Table 4.3 shows the trend for the size of the cube set when incrementally imposing four linear constraints on two 12-bit variables. The number of cubes generated by the customized software that targets optimizing the size of the cube set is generally shown to be slightly smaller than by the flow based on hardware synthesis

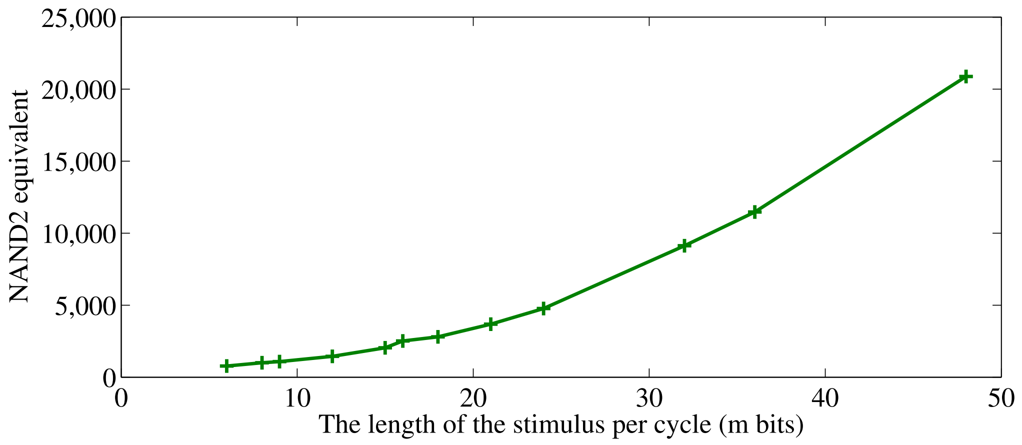


Figure 4.12: The hardware cost of the proposed CRSG for sequential constraints, according to the length of stimulus per cycle, in which the number of decoders is adapted to the length of stimulus, i.e., $\lceil m/3 \rceil$, which supports any case no matter the compaction efficiency.

and BDD-based generation of cubes.

It is important to note that the customized algorithms for method (a) from Table 4.3 need to use a two-level logic minimizer as a post-processing step. While this reduces the cardinality of the cube set, the reliance on a two-decade old public-domain tool, i.e., Espresso (McGeer *et al.*, 1993), leads to runtimes on the order of hundreds of seconds for the use cases from Table 4.3. On the other hand, the results for method (b) from Table 4.3 do not use a two-level logic minimizer for post-processing the cube set. These results have been obtained using a modern commercial hardware synthesis tool (Synopsys, 2016), as well as a state-of-the-art BDD package (Somenzi, 2012), and therefore its runtimes are on the order of a few seconds (up to low tens of seconds). Exploratory experiments that have used a two-level logic minimizer to the cube sets produced by the BDD-based method, have confirmed that the number of cubes will become nearly identical to the ones obtained by method (a), nevertheless

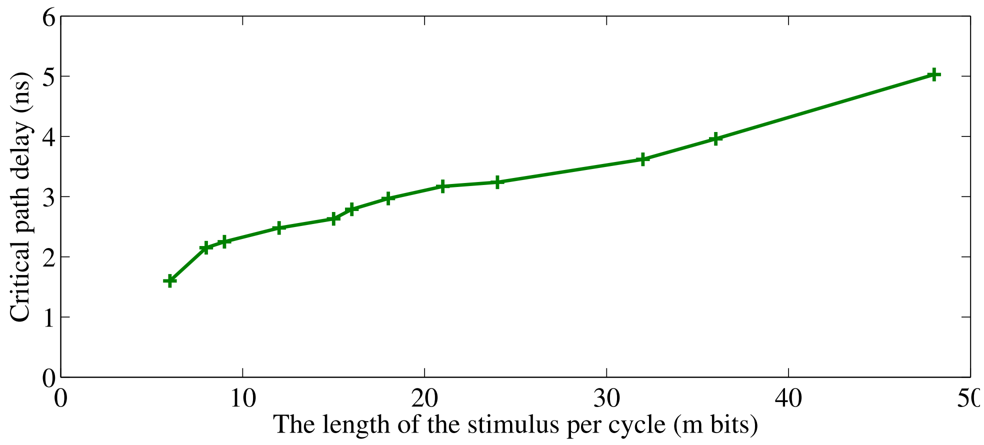


Figure 4.13: The critical path delay of the proposed CRSG for sequential constraints according to the length of stimulus per cycle.

the runtimes will increase. In the following, although the method (b) is more scalable in terms of runtimes, the number of optimized cubes that are generated by method (a) are reported. Overall, the outcome of this experiment confirms the scalability of the method according to the number of constraints. That is, it can be observed that the number of cubes goes down when more constraints are imposed incrementally because the valid sample space can be reduced (as illustrated in Figure 4.14):

1. If only the first constraint in the table is used, then the total number of valid

Table 4.3: The trend for the size of the cube set generated by (a) the customized software and (b) the indirect flow using BDD, when adding constraints incrementally for the ILP inequalities on 12-bit variables.

Constraints	**Valid pairs**	**Cubes by (a)**	**Cubes by (b)**
(1) $1000 \leq x + 2y \leq 8000$	11943292	5724	5768
(2) $y \geq 5x - 6000$, and (1)	5243774	4482	4528
(3) $x \geq 600$, and (1),(2)	3143474	3398	3474
(4) $y \geq 2000$, and (1),(2),(3)	1582674	2120	2161

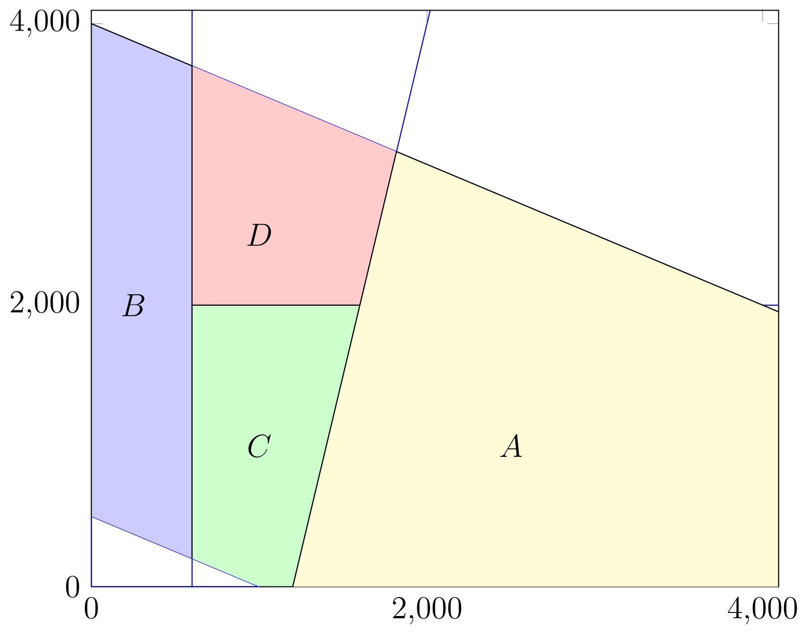


Figure 4.14: The geometrical illustration for the incrementally imposed constraints from Table 4.3.

pairs is about 1.2×10^7 and the number of cubes is 5724; from the geometrical standpoint as illustrated in Figure 4.14, this constraint covers all the four regions, i.e., $A \cup B \cup C \cup D$;

2. If the second constraint is used (in addition to the first one), then region A will be eliminated and the number of cubes becomes 4482;
3. If the third constraint is added, region B is eliminated and the number of cubes is 3398. The case for the forth constraint is similar.

What can be observed is that while the valid space gets constrained and the valid stimuli are "close" ("close" in the Boolean domain, i.e., implicants/cubes can be generated that cover a large number of minterms/valid stimuli) to each other, the cube count will go down when more constraints are added. It demonstrates for this type of problems that the cube count tends to go down when more constraints are added, mainly because the valid pairs count is reduced.

In order to evaluate the effectiveness in reducing the storage requirements for large validation sequences, the CRSGs are configured to generate stimuli for resembling 168-bit packet heads for H.264 real-time transport protocol (RTP) (Wang *et al.*, 2011) shown in Figure 4.15, as well as 160-bit packet heads in the PCI-express (PCIe) 3.0 transaction layer packet (TLP) format (Ajanovic, 2009) shown in Figure 4.16. Each field in the packet head must satisfy the requirements specified in the protocol standards, including the format, defined/reserved values and the coordination among fields. The fields that can be randomized are extracted for the design of constraints, thus leaving the non-random CRC field to be attached by CRC computation logic. The results of converting the constraints to cubes with the customized software tools in Section 3.3, are listed in Table 4.4(a). In addition to supporting logic constraints, i.e., $T = 1$, the method for mixed type constraints can generate each packet in multiple cycles as a partitioned packet. Table 4.4(b) illustrates the CBC set size for different values of T. When T increases, the length of the stimulus generated in each cycle decreases, which leads to fewer byte decoders and consequently less hardware overhead.

Only the cubes in the CBC format are required to be loaded to the cube memory; this requires a quarter to a half of the storage needed for the binary cubes. Compared to the reference method (Kinsman *et al.*, 2013) that stores basis vectors from which LFSR seeds are expanded on-the-fly, the volume of data that is required by the proposed method is at least an order of magnitude less. This is because the number of basis vectors from (Kinsman *et al.*, 2013) needed to satisfy a particular cube can be

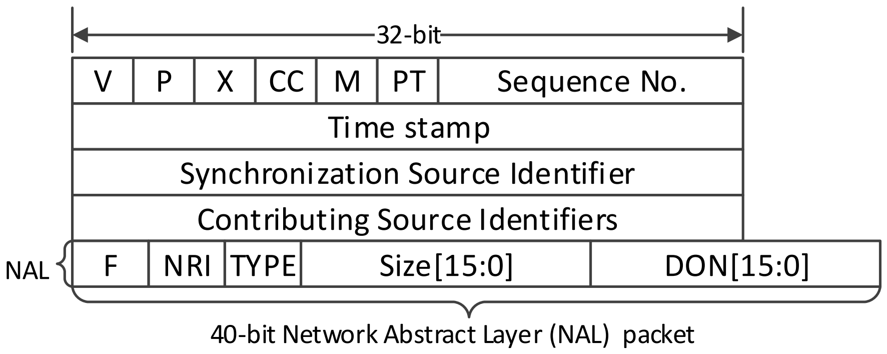

32-bit
V
P
X
CC
M
PT
Sequence No.
Time stamp
Synchronization Source Identifier
Contributing Source Identifiers
NAL
F
NRI
TYPE
Size[15:0]
DON[15:0]
40-bit Network Abstract Layer (NAL) packet

Figure 4.15: A typical packet format of H.264 RTP.

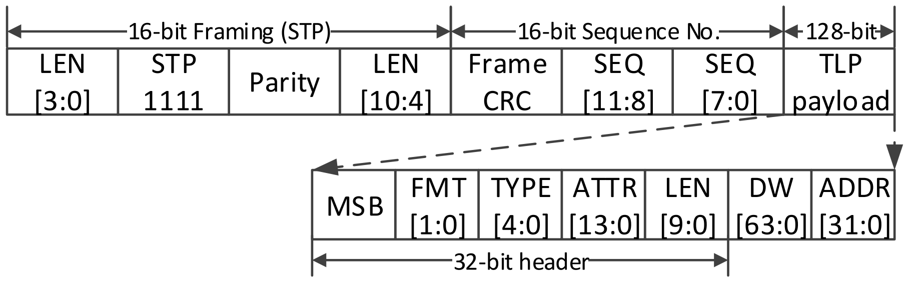


Figure 4.16: A typical packet format of PCIe 3.0 TLP.

large and, more importantly, the dimension of each of these vectors is as large as the LFSR size. Hence, the savings of the proposed CRSG are explained by the fact that the storage requirements are not dependent on the LFSR size. Table 4.5 shows the total size of data that are applied to the design for the case of H.264 RTP and PCIe TLP compared with the reference method (Kinsman *et al.*, 2013). Considering that the total number of constrained-random patterns that can be applied to the design using only one cube is defined by 2^{ξ}, where ξ is the number of free bits (i.e., 'X' bits) provided in Table 4.4(a), it is evident that the number of stimuli that can be used for validation can easily meet the objectives of real-time execution that lasts for hours.

It should be also noted that in the event that the capacity of the on-chip memory is

Table 4.4: The encoding result of the cube set for PCIe and H.264.

(a) The results by the algorithms for (1) logic constraints and (2) sequential constraints

Packet head format	**Free bits**	**Cube count**	**Binary cube set size**	**CBC set size by (1)**	**CBC set size by (2)**
H.264 RTP	158	335	14.91KB	3.9KB	3.2KB
PCIe TLP	127	5119	204.76KB	81.8KB	76.4KB

(b) The compaction results of CBC sets according to the generation period T

T	**Items**	**H.264 RTP**	**PCIe TLP**
2	CBC set size	3.6KB	81.5KB
	Stimulus length m	84b	80b
4	CBC set size	3.8KB	83.5KB
	Stimulus length m	42b	40b

a tight implementation constraint (e.g., approximately 80 KB for PCIe TLP might be excessive for some designs), one can update CBCs on-chip dynamically, as described in Section 4.2. The main reason why this dynamic update is feasible is because *any* CBC will be decoded into a valid mask that will ensure that the pseudo-random patterns at the output of the LFSR will be mapped onto functionally-compliant stimuli. This is a direct consequence of translating the SystemVerilog constraints into cubes, as described in Section 3.1. For example, considering that 5,119 CBCs for PCIe TLP in Table 4.4(a) require 81.8 KB, one can store approx 250 CBCs into a 4 KB memory block; in such a memory-constrained environment, the CRSG can iterate through the masks expanded by these 250 CBCs, while a new subset of CBC is loaded through a low-bandwidth serial interface from on-board storage or directly from the host.

Table 4.5: Loaded data volume and the total number of implied stimuli, compared with the reference method(Kinsman *et al.*, 2013).

(a) Generating H.264 RTP

Methods	**Loaded data type**	**Loaded data size**	**No. of stimuli**
Reference	seeds for 24-bit LFSR	182Kb	1.23×10^9
	seeds for 128-bit LFSR	5.41Mb	2.50×10^{40}
Proposed	CBC for logic constraints	31.2Kb	1.22×10^{50}
	CBC for sequential constraints	25.6Kb	

(b) Generating PCIe TLP

Methods	**Loaded data type**	**Loaded data size**	**No. of stimuli**
Reference	seeds for 24-bit LFSR	2.46Mb	1.69×10^9
	seeds for 128-bit LFSR	80.9Mb	3.26×10^{40}
Proposed	CBC for logic constraints	0.654Mb	8.71×10^{41}
	CBC for sequential constraints	0.611Mb	

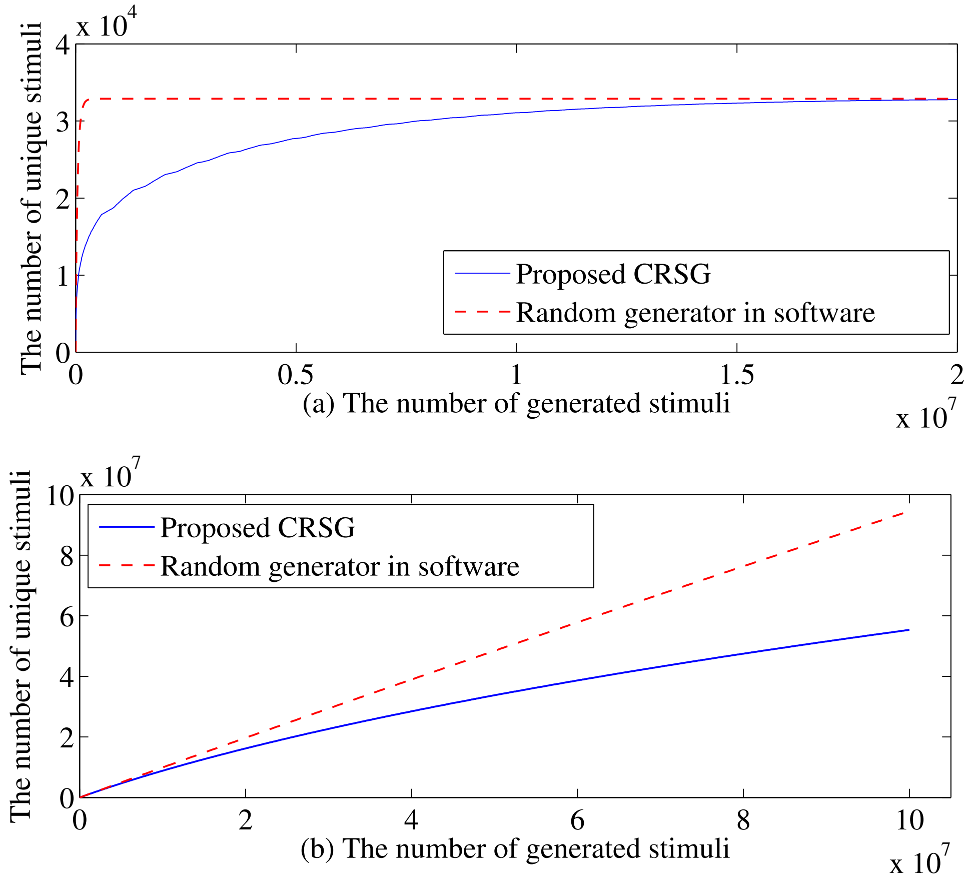


Figure 4.17: The relation between the number of generated stimuli and the number of unique stimuli based on the constraint $x \geq y$. The unsigned variables x and y are set to 8 bits in (a) and 16 bits in (b) respectively.

4.4.4 The evaluation of stimuli distribution

Concerning the quality of the randomized stimuli generated by the proposed solutions, the distributions of the stimuli produced by CRSGs are examined.

Figure 4.17(a) illustrates the relation between the number of stimuli generated and the number of unique stimuli based on a simple constraint $x \geq y$, in which x and y are unsigned 8-bit variables. If the random values are drawn from the uniform distribution then, until the entire valid space (as defined by the constraint) is exhausted, the value on the vertical axis should almost match the value on the horizontal axis; thereafter, the value on the vertical axis saturates to the maximum number of unique stimuli. The maximum number of valid pairs that satisfy $x \geq y$ is 32,896 and the software-based random generator in a SystemVerilog compliant simulator (Synopsys, 2015) reaches this saturation point after 543,085 patterns; it should be noted that these results have been obtained using the **rand** type for the randomized variables for the software random generator). As the number of generated patterns increases, the number of unique patterns generated by the hardware method is approximately half of the ones generated in a software simulator (for the same number of total patterns). Figure 4.17(b) shows this trend more clearly, where x and y are set to be 16 bits each. It should be noted that during post-silicon validation it is reasonable to have experiments with a significantly larger number of clock cycles than during pre-silicon verification; therefore, though the valid randomized stimuli are repeated more often in the hardware implementation, as confirmed by this experiment, the extensive number of clock cycles exercised on silicon prototypes, which is at least four orders of magnitude more than during pre-silicon simulation (Goodenough and Aitken, 2010), are expected to compensate for this repetition of constrained-random stimuli.

4.5 Summary

In this chapter the solution for generating stimuli according to logic constraints has been first introduced. In particular, this solution relies on a compaction strategy which facilitates the design of cost-efficient on-chip decoders. Then in order to generate stimuli based on sequential constraints, an extended solution has been proposed. The solution not only expands the support for sequential constraints, but it also increases the speed of processing each cube (as fast as one clock cycle per cube). The experimental results have investigated the impact of the proposed solutions on the area cost, clock speed, volume of data and the quality of the constrained stimuli in terms of their repetition.

When the distribution of stimuli is of critical importance during post-silicon validation, methods to avoid the repetition of stimuli will be discussed in detail in the next chapter.

Chapter 5

Controlling the distribution of the constrained-random stimuli

The solutions in Chapter 4 are focused on the features of functional constraints. In this chapter, the characteristics for the distribution of on-chip stimuli generation are investigated and improved, followed by a series of solutions supporting the features compliant to the SystemVerilog standard.

The root causes of repetitive stimuli are first examined. As elaborated in this chapter, the repetition can be caused by either the overlaps between cubes or sampling the same pattern multiple times from the same cube. The first problem is addressed by proposing an efficient algorithmic procedure for reshuffling the cubes in order to ensure the same constrained space is preserved (with insignificant overhead in cube count) and also guaranteeing that the intersection between any two cubes is void. The second problem is addressed through a new on-chip generator with a *dynamic* LFSR structure and a vector assembler, which ensure that, as constrained patterns are generated on-the-fly from any given cube, they are never repeated. Hence the solution can generate in real-time uniformly distributed (more specifically, *random-cyclic*) stimuli with no repetition until the space of the compliant stimuli is exhausted.

Furthermore, the extended generator is proposed which introduces a scheduler for cubes in order to avoid consecutive patterns to be constrained by the same cube. Not only that it can ensure that any compliant pattern is not repeated until the entire constrained space is enumerated, but it can also reprogram the cube schedule from one enumeration to another. Moreover, in order to support weighted distributions as specified in SystemVerilog, a solution that can deal with multiple sets of cubes is presented.

5.1 The motivation for controlling the distribution during stimuli generation

The results from Section 4.4 showed that the unique patterns applied in-system are approximately half of the ones generated by a software simulator. Since in-system validation runs significantly faster than simulation, it is acceptable in practice that the quality loss due to stimuli repetition is compensated by the quantity of clock cycles and patterns applied at the post-silicon phase (Adir *et al.*, 2011). Nonetheless, if the redundant cycles due to repeated stimuli can be reduced, the saved cycles can be allocated to previously unused compliant stimuli that can stress the DUV into valid states which otherwise could not have been explored. For this reason, for pre-silicon verification SystemVerilog standardizes the features of random modifiers. For example, while both **rand** and **randc** ensure that the random variables are sampled from a uniform distribution in the constrained space, only **randc** guarantees that a random variable does not get assigned the same value twice until the entire constrained space has been exhaustively enumerated (referred to as *random-cyclic* stimuli generation).

The contribution from this chapter is driven by the following question: what would be the cost, in terms of content preparation and on-chip generation (both storage and area), to support the application of *random-cyclic* stimuli in hardware? To answer this question, an enhanced flow is developed, including preparing user-programmable content for on-chip CRSG, as well as new circuit blocks for on-chip stimuli generation. The results in Section 5.7 indicate that, with similar area and storage as prior works, uniformly distributed stimuli can in fact be applied in post-silicon validation environments. Based on this method, further work is done to refine the evenness of the distribution by interleaving cubes during generation, as well as supporting the weighted distributions compliant to SystemVerilog.

5.2 Causes of stimuli repetition

Before developing the software and hardware blocks needed to generate uniformly distributed constrained-random patterns in real-time, it is critical to develop the insights for the causes that lead to stimuli repetition.

One cause for the repetition of stimuli is the *overlap* between the stimuli implied by two or *multiple* cubes. For example, consider two cubes "1X0X" and "11XX" from the set of cubes derived from the constraint $x \geq y$ shown in Code 2.1. As illustrated in Figure 5.1(a), both cubes imply the binary vector "1100" (and "1101" as well), so the CRSG hardware will generate "1100" twice if both cubes are loaded. To address this problem, a new algorithm based on the mathematical model developed in Section 3.2 will be elaborated, which rectifies the overlapped cubes without any loss of the valid pattern space.

The other cause for repetition stems from the interaction between the autonomous sequence generated by the LFSR and the on-chip stimuli correction strategy. The unconstrained sequences from the output of the LFSR serve as the input to the correction logic. This correction logic essentially forces the specified bits from a cube and lets the don't-care bits to be filled with random values from the LFSR. As shown in Figure 5.1(b), given a cube "1X0X" and a 4-bit unconstrained sequences {. . . , 1101, 1110, 1111, 0111}, the correction logic would mask the left first bit with '1' and the third bit with '0'. Hence, the stimuli sequence will be {. . . , 1101, 1100, 1101, 1101}. As a result, although all the masked (corrected) stimuli are made compliant to the constraint (based on cube "1X0X"), pattern "1101" is generated multiple times because the LFSR values for the 'X' positions will happen to be the same for multiple clock cycles. To address this problem, the new hardware generator including a dynamic LFSR and a flexible correction component (vector assembler) that can avoid repetition for the *same* cube are designed, as elaborated in Section 5.4.

5.3 Generate non-overlapped cubes

The flow for cube processing is similar to the general flow shown in Section 3.3. The constraints on stimuli for validation sessions can be formalized using SystemVerilog. The proposed method first converts the constraints into the set of *non-overlapped* cubes. Then the set of non-overlapped cubes are encoded into a binary format, which can be loaded on-chip.

This section presents a new algorithm used to *rectify* a set of cubes in order to ensure that any two patterns expanded from two distinct cubes will not be identical.

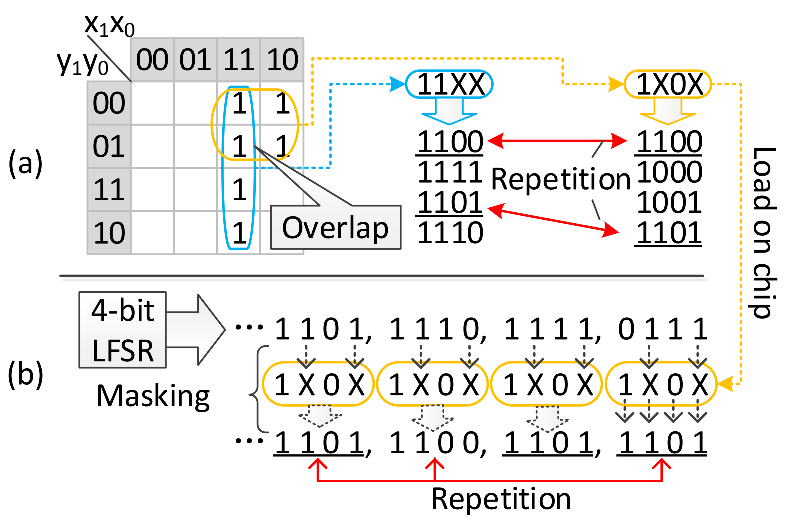


Figure 5.1: Stimuli repetition due to (a) overlapped cubes and (b) LFSR and correction strategy.

Section 3.3 has shown the general algorithms for converting the constraints into equivalent cubes and encoding them into binary format. First the constraints in SystemVerilog are converted with manually-guided customized software and then a third-party logic minimization tool can be used. The software and the tool can deduce and enumerate compliant cubes according to the constraints' types (such as, for example, arithmetic, logic or conditional expressions). They also use heuristic algorithms to merge cubes and decrease the size of the set of equivalent cubes. However, the overlapped cubes can be introduced during this step. In particular, a two-level logic minimization tool, such as Espresso (McGeer *et al.*, 1993), is designed for minimizing the number of cubes *and* specified bits (i.e., the non-'X' bits) in a cube; hence, due to its original objective (i.e., two-level logic minimization), it is not concerned with recognizing and removing overlaps between cubes. For instance, if the customized processing software produces a set of cubes {110X, 100X, 111X} for an arbitrary constraint, the tool then deduces the minimized equivalent set as {11XX, 1X0X}. Note, this result set contains overlapped cubes. As a consequence, an add-on algorithm to process the cubes produced by a logic minimizer is developed, as shown in Code 5.1. It rectifies the overlapped cubes *without any loss of the patterns that can be implied.* Furthermore, this algorithm also maintains the goal to limit the total number of rectified cubes.

Based on the definitions from Section 3.2, some corollaries on the cube pair $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ are introduced, which are needed to explain the algorithm for rectifying the set of cubes to avoid overlaps between cubes.

- If $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ is a mutually exclusive cube pair, $\boldsymbol{b}$ does not overlap with $\boldsymbol{a}$.
- If $\boldsymbol{b}$ is a sub-cube of $\boldsymbol{a}$, $\boldsymbol{b}$ is redundant in the set and removable as long as $\boldsymbol{a}$ is in the same set.
- Otherwise, i.e., there is no mutually exclusive bit pair but at least one overlapped bit pair in them, then $\boldsymbol{b} \cap \boldsymbol{a} \neq \emptyset$. Hence $\boldsymbol{b}$ overlaps with $\boldsymbol{a}$ and it needs to be rectified according to the reference cube $\boldsymbol{a}$.

Code 5.1 iterates in a given cube set s_{in} and produces the equivalent rectified set s_{out}, in which any two cubes are mutually exclusive. The function RECTIFYSET first *deliberately* chooses a reference cube $\boldsymbol{a}$ out of s_{in}, and finds out the overlapped pairs with it. For each overlapped pair $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$, the sub-function RECTIFYCUBE produces the rectified result of $\boldsymbol{b}$ according to $\boldsymbol{a}$, which is ready to take the place of $\boldsymbol{b}$ in s_{in} for the next reference cube. After processing all overlapped cubes with $\boldsymbol{a}$, it indicates that no cubes in s_{in} overlaps with $\boldsymbol{a}$. Hence $\boldsymbol{a}$ is moved from s_{in} to s_{out}, and s_{in} is updated for the new reference cube in the next iteration. Figure 5.2 shows the rectified result of s_{in} ={1X0X,11XX}. Compared with Figure 5.1(a), the overlapped

Code 5.1 The algorithm to rectify overlapped cubes for the given set s_{in} and produce the equivalent set s_{out}.

```
function RectifySet($s_{in}$)
    $s_{out} \leftarrow \emptyset$
    Compute initial values of $\rho_0, \rho_1, \cdots, \rho_{m-1}$
    while $s_{in} \neq \emptyset$ do
        $\boldsymbol{a} \leftarrow$ FindReferenceCube($s_{in}$)
        for each cube pair $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ in $s_{in}$ do
            $s_{in} \leftarrow s_{in} - \{\boldsymbol{b}\}$
            $s_{next} \leftarrow s_{next} \cup$RectifyCube($\boldsymbol{a}$,$\boldsymbol{b}$)
        end for
        $s_{out} \leftarrow s_{out} \cup \{\boldsymbol{a}\}$
        $s_{in} \leftarrow s_{in} \cup s_{next}$
        Update $\rho_i$ on all relative positions $i$
    end while
    return $s_{out}$
end function

function RectifyCube($\boldsymbol{a}$,$\boldsymbol{b}$)
    if $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ is a mutually exclusive pair then
        return $\{\boldsymbol{b}\}$
    end if
    $s_c \leftarrow \emptyset$                                  ▷ The result set of $\boldsymbol{b} - \boldsymbol{a}$
    $s_{pairs} \leftarrow$ all overlapped bit pairs in $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$
    sort $s_{pairs}$ by $\rho$ from small to large
    for each $\langle a_i, b_i \rangle$ in $s_{pairs}$ do
        $b_i \leftarrow \overline{a_i}$                          ▷ Halve $\boldsymbol{b}$
        $s_c \leftarrow s_c \cup \{\boldsymbol{b}\}$             ▷ The part of $\boldsymbol{b}$ in $\boldsymbol{b} - \boldsymbol{a}$
        $b_i \leftarrow a_i$
                            ▷ The other part of $\boldsymbol{b}$ for the next iteration
    end for
    return $s_c$
end function
```

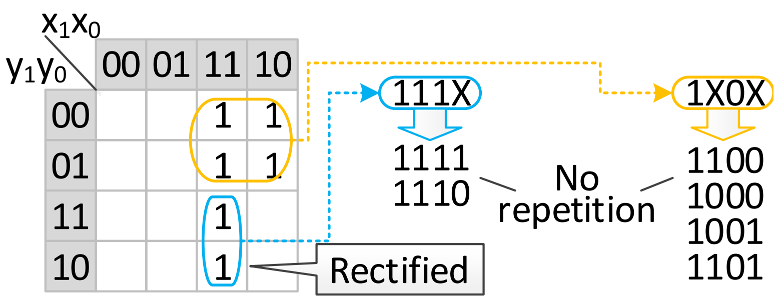


Figure 5.2: Eliminate repetition by rectifying overlapped cubes.

cube “11XX” is rectified as “111X” which is no longer overlapped with the reference cube “1X0X”. Meanwhile the result set s_{out} ={1X0X,111X} implies the same space of valid patterns as s_{in}.

The result of the function RECTIFYCUBE is the relative of complement of $\boldsymbol{a}$ in $\boldsymbol{b}$, i.e., $\boldsymbol{b} - \boldsymbol{a}$. As mentioned in Section 3.1, not all elements in the universal set 2^U can be equivalently expressed as one cube. Hence the result is a set containing one or several cubes. The cubes in the result set should imply all the binary vectors which are implied by $\boldsymbol{b}$ but not by $\boldsymbol{a}$. Note that, if $\boldsymbol{b} \subseteq \boldsymbol{a}$, the function produces $\emptyset$, so it simply removes the redundant cube $\boldsymbol{b}$ from s_{in}. The algorithm guarantees that all the cubes in the result set are mutually exclusive to each other, thus ensuring that the same pattern is not sampled from two different cubes.

In order to achieve the goal of lowering the total number of final cubes and specified bits, function FINDREFERENCECUBE uses cube weight introduced in Section 3.2 to choose the reference cube and calculating $\boldsymbol{b} - \boldsymbol{a}$.

Function FINDREFERENCECUBE calculates cubes’ weights using Equation (3.4). Then the heaviest cube is selected as the reference cube in each iteration. The cubes of a larger implied space (as measured by the first component in the equation) are preferred, because the algorithm then may use fewer cubes to cover the whole implied space of s_{in}. The adoption of X-density ρ in the second component quantifies the correlation with other cubes in the set, which reflects the probability of overlapping with other cubes. Selecting a cube with high correlation as the reference cube may result in more other cubes to be rectified, which could adversely increase the cube count. By setting $W_X = m$, the function FINDREFERENCECUBE will adopt a large implied space first strategy to select the reference cube for each iteration.

Function RECTIFYCUBE constructs a series of mutually exclusive cubes to reconstruct the vector space of $\boldsymbol{b} - \boldsymbol{a}$, in which $\boldsymbol{a}$ is the reference cube and $\boldsymbol{b}$ is the cube to be rectified. After removing the overlapped part with $\boldsymbol{a}$, the residual space of $\boldsymbol{b}$ (i.e., $\boldsymbol{b} - \boldsymbol{a}$) may not remain in regular cubic shape, i.e., it could not be a single cube. However, it can be constructed by a series of non-overlapped smaller cubic spaces. The function iteratively applies Shannon's expansion on $\boldsymbol{b}$ at overlapped positions i of $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ as:

$$\begin{aligned} f(b_i, b_0, b_1, \cdots, b_{m-1}) = & b_i f(1, b_0, b_1, \cdots, b_{m-1}) + \\ & \overline{b_i} f(0, b_0, b_1, \cdots, b_{m-1}) \end{aligned} \tag{5.1}$$

It adopts greedy searching for the smallest ρ_i during enumeration to reduce the number of cubes in the result set. The property of mutual exclusion among cubes is guaranteed by constructing mutually exclusive bit pairs in them. The total number of produced cubes for $\boldsymbol{b} - \boldsymbol{a}$ is $\sum_{i=0}^{m-1} \mathcal{X}(b_i)(1 - \mathcal{X}(a_i))$, i.e., the number of overlapped bit pairs of $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$. The time complexity for the worst case is $O(m \times log_2 m)$ decided by the sorting procedure (Code 5.1 line 23).

The time complexity of the whole algorithm depends on the size of s_{in}, which is dynamically updated according to the reference cube selection and the result of cube rectification. The experimental results in Section 5.7 show that the heuristic algorithm can tackle constraints with thousands of cubes in a few seconds.

It is also worth mentioning that, to reduce the final number of cubes, an optional merging step can be performed on s_{out} by Espresso (McGeer *et al.*, 1993). It performs a quick distance-1 merge on the set. It can be seen as the inverse process of halving a cube. That is, during each iteration, it selects a cube pair $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ out of s_{out} which holds the same values ('0','1' or 'X') on the same position except for only one mutually exclusive pair $\langle a_i, b_i \rangle$. Then it modifies $a_i = X$, and then only places the modified $\boldsymbol{a}$ back to s_{out} for the next iteration. Hence it reduces the cube count by one, while the properties of mutual exclusion for non-overlapping cubes are preserved. The time complexity of this step is $O(|s_{out}| \times \log_2 |s_{out}|)$, as reported in (McGeer *et al.*, 1993). For instance, the cube "111X" and "110X" can be merged into "11XX". It should be noted that one has to run Espresso with `-Dd1merge` to ensure that only the above-mentioned step is performed.

The above rectification process follows the flow shown in Section 3.3, where the generated cubes are encoded into the binary format and could be optionally compacted using the loose coupling compaction algorithm presented in Section 4.3.

5.4 The on-chip generator for random-cyclic stimuli generation

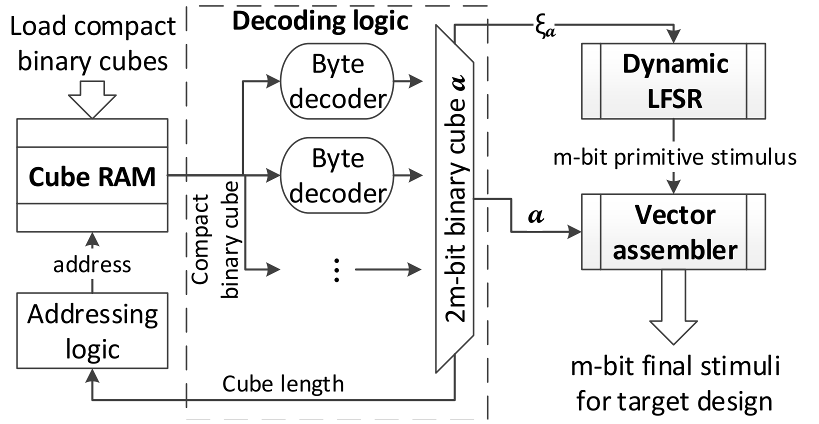
As for all the variants of the on-chip CRSG hardware from this thesis, the CRSG for random-cyclic stimuli generation receives the equivalent set of cubes from user-defined constraints and forces the LFSR sequences to be functionally-compliant by employing cubes as masks. As shown in Figure 5.3(a), the structure consists of embedded cube RAM, decoding logic, a customized dynamic LFSR and a vector assembler.

The binary cubes, or the compact binary cubes (CBCs) if compaction is employed, are loaded onto the embedded RAM. (CBCs, if applicable, are decoded to binary cubes on-chip.) Then the dynamic LFSR *and* the vector assembler check the cubes in order to generate the binary sequences of compliant stimuli. In the following, the steps of cube decoding and compliant stimuli generation are elaborated. It should be noted that the left hand side of Figure 5.3(a) builds on the architecture shown in Figure 4.6 and therefore most of the details from this section are provided for the new blocks from Figure 5.3(b) and Figure 5.3(c) respectively, which are critical to avoid the repetition of patterns that are constrained by the same cube.

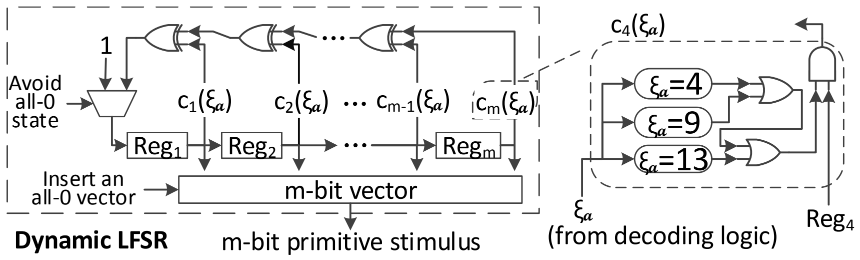
5.4.1 Decode cubes on-chip

The on-chip embedded cube RAM stores binary cubes (or CBCs) that have been transmitted, for example, via a low-bandwidth interface. It only requires small capacity for loading a few cubes instead of the entire cube set. The addressing logic can record the status of cubes and overwrite the used cubes by new cubes that can be uploaded after all the patterns from the initial cube set have been exhausted. That is, the cube RAM serves as a fist-in-first-out cube queue and refreshes its content as the validation process progresses.

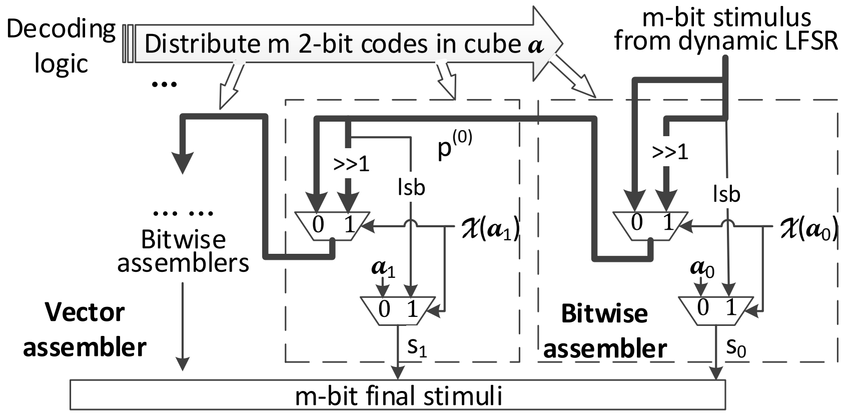
The decoding logic is similar to the one in Figure 4.6. It consists of a series of combinational byte-wise decoders, which decode the CBCs from the cube RAM to binary cubes. Each byte decoder processes one byte in the CBC. The results from the byte decoders are combined together, which form a complete binary cube for stimuli



(a) The top level of CRSG hardware



(b) The inner structure of dynamic LFSR.The logic of $c_4(\xi_a)$ exemplifies the typical logic for switching functions $c_i(\xi_a)$



(c) The inner structure of vector assembler, which comprises m bitwise assemblers

Figure 5.3: The structure of on-chip CRSG hardware for uniformly distributed stimuli generation.

generation in the vector assembler. Meanwhile, the length of the CBC is sent back to the addressing logic for computing the address of the next cube. Also the attribute ξ of the cube is sent to the dynamic LFSR, which is used as explained in the next subsection. It is worth noting that if a user chooses to transfer binary cubes directly on-chip there is no need for the decoding logic (i.e., for applications where on-chip area is more important than on-chip storage), however attribute ξ of each cube still needs to be sent to the dynamic LFSR.

5.4.2 Generate ξ-bit primitive sequences

The m-bit LFSR, as used in the previous solutions in Chapter 4, supplies the m-bit primitive stimuli, which is altered according to the content of cubes. However, such fixed-length LFSR lacks the flexibility of varying distribution of the sequences according to cubes, which causes stimuli repetition. Therefore, a dynamic LFSR is designed based on it, and ξ, i.e., the number of 'X's in the cube (defined in Equation 3.2 from Section 3.2), is selected to control the behaviour of the dynamic LFSR as a degenerated ξ-bit LFSR, as detailed next.

An m-bit LFSR consists of m 1-bit registers and `XOR` gates in its feedback. The behaviour can be modelled with the characteristic polynomial $f_m(x) = 1 + c_1x + c_2x^2 + \cdots + c_mx^m$. If the coefficient $c_i \neq 0$ $(1 \leq i \leq m)$, it indicates the ith register is used for the feedback computation. Note if $f(x)$ is a primitive polynomial, the LFSR can enumerate non-repetitive $2^m - 1$ vectors of m bits except the all-0 vector (the forbidden state of LFSR) within each period $T = 2^m - 1$. Note, the theory of primitive characteristic polynomials over Galois Fields and how they relate to LFSR models can be found in Bardell *et al.* (1987).

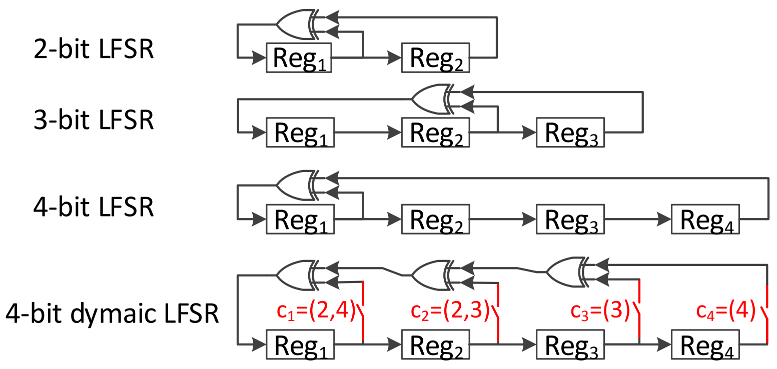


Figure 5.4: The sketch of 4-bit dynamic LFSR evolving from fixed-length LFSRs, which makes c_i as a switching function based on the on-set.

The proposed dynamic LFSR changes c_i from a constant coefficient to a variable function that is dependent on ξ. Figure 5.4 depicts the framework of 4-bit dynamic LFSR, which can degenerate to 2/3/4-bit fixed-length LFSR. That is, the feedback with the ith register depends on ξ. If and only if $c_i(\xi) \neq 0$, the ith register is connected to the feedback net. In particular, if $c_m = c_{m-1} = \cdots = c_{m-k+1} = 0$ and $c_{m-k} \neq 0$, the m-bit LFSR is degenerated to $(m-k)$-bit LFSR. The switching functions are designed based on the binary primitive polynomials table (Živkovic, 1994), so that the LFSR can dynamically degenerate to ξ-bit long, while at the same time keeping $f_\xi(x)$ as a primitive polynomial. Table 5.1 exemplifies the switching functions for up-to 16-bit long LFSR. A complete listing of switching functions for up to 64-bit dynamic LFSR is given in Appendix A. For example, $c_4 = 1 \iff \xi \in \{4, 9, 13\}$, while $c_8 = 1 \iff \xi = 8$.

The left part in Figure 5.3(b) shows the general structure of the dynamic LFSR. Each bitwise register is connected to the logic which implements $c_i(\xi)$. The right part in Figure 5.3(b) illustrates a typical structure of the switching logic, including a series of comparators connected with `OR`-gates and an `AND`-gate. The number of

Table 5.1: Switching functions for a 16-bit dynamic LFSR that can be configured into smaller LFSRs

Functions	On-set	Functions	On-set
c_1	1,2,4,6,7,8,13,14	c_9	9
c_2	2,3,5,11,16	c_{10}	10
c_3	3,10,12,13,16	c_{11}	11,14
c_4	4,9,13	c_{12}	12,14
c_5	5,8,16	c_{13}	13
c_6	6,8	c_{14}	14
c_7	7,12	c_{15}	15
c_8	8	c_{16}	16

the comparators and the operands are set based on Table 5.1. For instance, because the on-set of $c_4(\xi)$ is {4,9,13}, the logic for $c_4(\xi)$ includes 3 comparators (against constants), which compare ξ with 4, 9 and 13. In addition, the first bitwise register is set to 1 during cube switching to avoid the forbidden all-0 state, and an all-0 vector generator is used for adding an all-0 vector to the output sequences. Consequently, the proposed dynamic LFSR can enumerate non-repetitive 2^ξ vectors in the first ξ bits of the output in consecutive 2^ξ cycles. This helps eliminate the problem of repetitive patterns within each cube. For example, compared with the fixed length LFSR in Figure 5.1(b), the dynamic LFSR shown in Figure 5.5 degenerates to 2-bit LFSR for the cube "1X0X" and it exhaustively enumerates four 2-bit primitive stimuli.

5.4.3 Assemble compliant stimuli

The vector assembler maps the 2^ξ vectors from the LFSR to the corresponding bit positions in the stimulus that is applied to the target design. As shown in Figure 5.3(c),

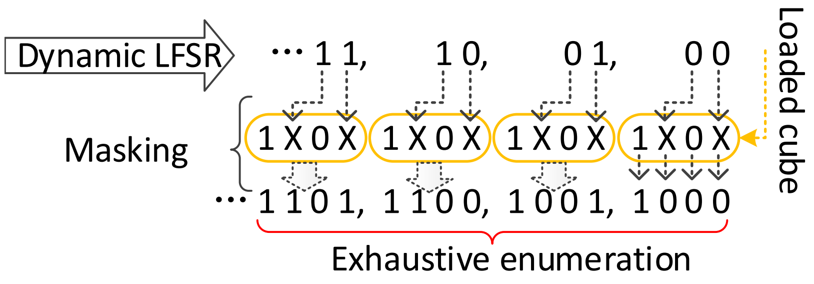


Figure 5.5: Exhaustive enumeration by dynamic LFSR and the new correction strategy, compared with Figure 5.1(b).

it consists of m bitwise assemblers to produce a compliant m-bit stimulus s, based on the $2m$-bit binary cube $\boldsymbol{a}$ from the decoding logic and ξ_a-bit vector as a primitive stimulus from the dynamic LFSR. The ith bitwise assembler $(0 \leq i \leq m-1)$ receives a vector $p^{(i-1)}$ from the $(i-1)$th bitwise assembler (except that the 0th assembler receives the vector from the LFSR) and passes a modified vector $p^{(i)}$ according to $\mathcal{X}(a_i)$. Meanwhile it produces s_i as one bit in s based on $\mathcal{X}(a_i)$. ($\mathcal{X}(a_i)$ is exactly the higher bit for each value in the binary cube as explained in Section 3.1.) Using arbitrators, it determines the outputs according to $\mathcal{X}(a_i) = 1$ as:

- If $\mathcal{X}(a_i) = 0$, it means a_i is a specified constant '0' or '1'. Then $s_i = a_i$ and $p_{(i)} = p^{(i-1)}$.
- Otherwise $\mathcal{X}(a_i) = 1$. It means a_i is an unspecified bit 'X'. Then $s_i = p_0^{(i-1)}$ and $p^{(i)} = \langle p_{\xi_a-1}^{(i-1)} \cdots p_2^{(i-1)} p_1^{(i-1)} \rangle$. That is, it uses a random bit originated from the dynamic LFSR to fill a bit in the final stimulus, and passes the left random bits (right shifting $p^{(i-1)}$ by one bit) to the following assembler.

Hence, the vector assembler selects the first $\xi_{\boldsymbol{a}}$ bits from the LFSR and the specified bits in the cube a to assemble a final compliant stimulus. As the example shown in the correcting process of Figure 5.5, each bit of the primitive stimulus from the dynamic LFSR substitutes an 'X' in the cube one by one. Thus the vector assembler assembles a complete 4-bit stimulus with the 2-bit primitive stimulus and the 2 specified bits in the cube.

5.5 Interleaving cubes during on-chip generation

The previous sections were focused on the fundamentals for controlling distribution of on-chip stimuli generation, based on which the solution for generating uniformly-distributed constrained-random stimuli was presented. Nevertheless, it partitioned the entire compliant sample space into a series subspace expressed by cubes, and it random-cyclically sampled these subspaces one by one. In such way, the distribution of cyclic-randomness still holds for the entire compliant space, however consecutive samples are constrained by the same cube. In case that the generation process has to be stopped due to the time limit of on-chip validation, the latter cubes will never get the opportunity to be sampled. In such cases, the cubes that constrain the stimuli are not evenly scattered. This might be a concern because the excitations conditions might vary from one potential error to another, and it is desirable to interleave patterns that are expanded from every single cube.

In this section an enhanced solution is proposed, which builds on the method to control the stimuli distribution from the previous solution. It can not only generate cyclic-random distributed stimuli in the compliant sample space according to user-defined constraints, but it also alleviates the potential problems caused by an uneven sampling of cubes. This is achieved by introducing a cycle-sharing schedule for cubes. The hardware cost for the cube scheduler does not depend on the LFSR size, nor the amount of data that is applied per clock cycle.

5.5.1 The analysis of the order of cube processing

Since the compliant sample space is expressed by a set of cubes in the Boolean space, the stimuli are generated by forcing the known values from the cubes and filling in the don't-care bits with random values. This may lead to a sequence of stimuli that are clustered due to expansion of consecutive samples from the same cube.

Considering the constraint $x \geq y$ from Code 2.1, a sequence of stimuli generated based on the equivalent cube set {1X0X, 11XX, XX00, X10X, 1XX0} could be $\{\overbrace{1000, 1001, 1100, 1101}^{\text{Use cube 1X0X}}, \overbrace{1111, 1110, 1101, 1100}^{\text{Use cube 11XX}}, \cdots\}$. The sampling process enumerates all the stimuli based on one cube, followed by all the stimuli expanded from the following cube and so on. Therefore it concatenates the permutation specified by each cube to form a whole permutation specified by the constraint. Nevertheless, in case that the subspaces for the first few cubes are large and the validation has to halt due to the limited validation time, the latter cubes never get used. Consequently, assuming the cubes do not overlap, the stimuli represented by unused cubes will not be generated, thus raising concerns whether the corner states that get excited by stimuli expanded from the unused cubes get adequately exercised and validated. Consider also the example of Code 5.2 derived from the SystemVerilog standard document(IEE, 2013a), which generates addresses with constraints designed according to the specification for a valid bus packet. If the validation process had halted before the third cube in Table 5.2 started, no packets with addresses in the ProtectedMode (that are in the range from 8'h10 to 8'h3F) would be generated to validate the corresponding behavior.

In order to alleviate the potential deficiency caused by sampling one cube at a time, an on-chip cube scheduler will be presented next. The purpose of this scheduler

Code 5.2 The SystemVerilog constraints for generating bus packets.

```
typedef enum {
  RealMode, ProtectedMode, FullMode
} AddrMode;

class BusPacket;
  rand AddrMode mode;
  rand bit [7:0] addr;
  rand bit [7:0] data;

  constraint word_align {
    addr[0]==1'b0;
  }
  constraint addr_range {
    (mode==RealMode)      -> addr<=8'h0F;
    (mode==ProtectedMode) -> addr<=8'h3F;
  }
endclass
```

Table 5.2: The equivalent set of cubes for the constraints in Code 5.2.

`mode[1:0]`	`addr[7:0]`	`data[7:0]`	**Mode**
`0X`	`0000XXX0`	`XXXXXXXX`	RealMode, part of ProtectedMode
`10`	`XXXXXXX0`	`XXXXXXXX`	FullMode
`01`	`00XXXXX0`	`XXXXXXXX`	ProtectedMode

is to apply a time-sharing scheme by interleaving all the loaded cubes, in order to balance the generation of constrained-random stimuli from every single cube.

5.5.2 On-chip cube scheduling

The on-chip generator conceptually consists of three functional parts, i.e., the cube scheduler logic with the on-chip RAMs, the dynamic LFSR, and the vector assembler as shown in Figure 5.6. In particular, in addition to the cube RAM, as used in previous solutions, a context RAM is used as the second on-chip RAM for interleaving cubes. The cubes are preloaded to the on-chip cube RAM. The scheduler selects cubes based on a round-robin strategy. According to the selected cube, the vector assembler manipulates the random sequence from the dynamic LFSR and it assembles the final stimuli applied to the DUV.

Taking the hardware cost and efficiency into consideration, a compact on-chip cube scheduler is designed. The cubes are treated with equal priority, and they are interleaved within a T-cycle slice (if T equals 1 then cubes are switched every clock cycle). A cube is used until it expires (i.e., all the patterns that can be expanded from the respective cube have been generated). The scheduler also manages the context RAM. The fixed-length item in the RAM for each cube contains two fields for recording the accumulated cycles for the cube and the state of the LFSR respectively. The scheduler handles each cube as follows:

1. When a new cube is loaded, it is stored at the end of cube items in the cube RAM and it is waiting for its cycle slice; its context item is initialized to zero.

2. The scheduler circulates the cubes in the cube RAM. At the first cycle of each T-cycle slice, it sends a new cube to the vector assembler, and sends the corresponding LFSR state to the dynamic LFSR (the behaviour of the dynamic LFSR and the vector assembler is the same as for the ones used in Section 5.4). It also assigns cycles for the current slice according to ξ (the number of 'X's in the cube) and the number of patterns expanded from this cube so far t_{all} (this value is stored in the context RAM). If $t_{all} + T < 2^{\xi}$, it assigns T cycles; otherwise it assigns $2^{\xi} - t_{all}$ and the cube is marked to be expired at the end of the slice.

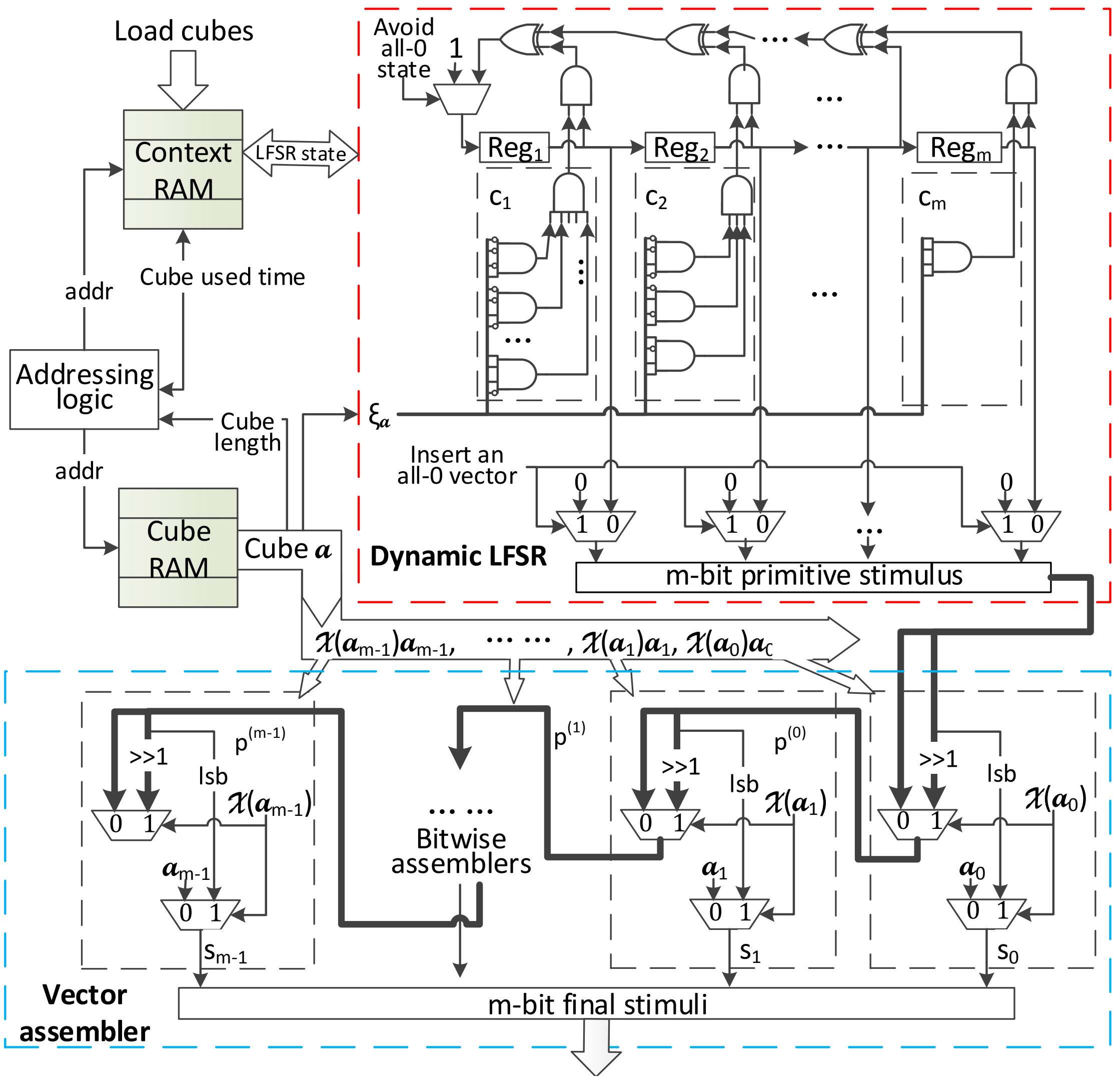


Figure 5.6: The structure of on-chip CRSG hardware to support cube scheduling.

3. If the cube does not expire, the cube is written back to the cube RAM at the address wb managed by the scheduler. Meanwhile the next state of the LFSR and the updated accumulated cycle count $t_{all} \leftarrow t_{all} + T$ are written back to the corresponding location in the context RAM, followed by incrementing wb. Note, writing the next state of the LFSR instead of the current state eliminates one stall cycle during the following T-cycle slice. Note also, if the cube expires, the scheduler only increments the counter for expired cubes (in the current circulation through all the cubes) without writing back. Since the cube RAM and the context RAM are dual-port RAMs, the scheduler also issues the read addresses (for both the cube and context RAMs) for the next cube.

4. When the current circulation ends at the last cube indicated by the address end, the scheduler updates end according to the expired cube counter, and then it resets this counter and wb in order to restart a new circulation.

Figure 5.7 illustrates a typical scheduling process. The scheduler manages to complete all the steps without stall cycles for switching cubes during stimuli generation, even if the cubes are switching cycle by cycle, i.e., $T = 1$. Meanwhile, the valid cubes are collected in the continuous address space for each circulation which eliminates extra cost for managing memory fragments due to expired cubes. This collection process is also done without extra clock cycles (by deciding not to increment wb for the expired cubes).

It is worth noting that, if the total number of cubes is very large, a subset of cubes could be buffered in the on-chip memory. Since the cube and context RAMs are both dual-port RAMs (and the write port is needed only at the end of a T-cycle or an expired time slice), the cubes that did not fit into the on-chip memory can be

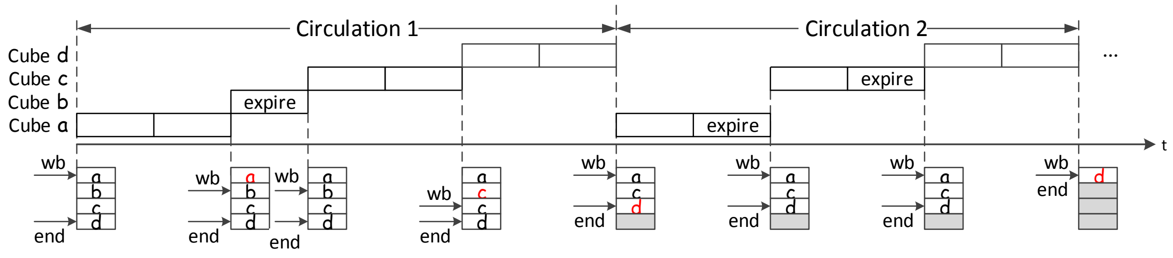


Figure 5.7: The timeline for scheduling cubes and the snapshots of the cube RAM with the address pointers *wb* and *end* ($T = 2$).

updated in the unused regions of these RAMs (marked in gray in Figure 5.7). After all the cubes have expired, one can reload the content of the on-chip memory with the cubes placed in a different order; also, the initial state of the dynamic LFSR for each cube can be changed when the cubes are reloaded. This will ensure that, in the case of very long validation sessions where the entire constrained space is exhausted and it needs to be enumerated again, in the second enumeration the order of the samples will be different.

5.6 Supporting weighted distributions

SystemVerilog supports sets of weighted distributions using the operator `dist`, as exemplified in Code 2.2. It allows additional constraints along with **dist** operations, which may influence the distribution. Implementations are not required to satisfy the distribution expressions if they are not satisfiable. Note, more details about weighted distributions can be found in the SystemVerilog standard (IEE, 2013a). The requirement for on-chip stimuli generation with weighted distributions introduces new circumstances that need to be supported, as elaborated in this section.

Code 5.3 Rewriting the **dist** expression in the constraint block from Code 2.2 to multiple constraint blocks using **inside** expressions.

```
class DistConstraint_sub1;
  rand bit [15:0] x;
  constraint x_dist_sub1 {
    x inside {[10:999]};
  }
endclass
```

(a) Assigned weight=3

```
class DistConstraint_sub2;
  rand bit [15:0] x;
  constraint x_dist_sub2 {
    x inside {[1000:2000]};
  }
endclass
```

(b) Assigned weight=2

5.6.1 Cube preparation for weighted ranges

In order deal with constraints with **dist** expressions, the cube preparation process disassembles **dist** expressions into a series of equivalent expressions with weighted ranges using **inside**. This step precedes the normal flow of cube conversion. Given a constraint block including **dist** expressions, each range in the expression is extracted one by one, which is combined with the other operations in the original constraint block to form an independent constraint block. Meanwhile, each new formed constraint block is assigned a weight according to the original expression. For instance, Code 5.3 shows two constraint blocks rewritten from the **dist** constraint in Code 2.2. Consequently, each rewritten constraint block could be converted to an equivalent set of cubes by the normal flow, including the conversion to non-overlapped cubes and their encoding into the binary format. Each set of cubes derives its weight from the correspondent constraint block. The on-chip generator uses these multiple sets of cubes, as explained in the next subsection.

5.6.2 On-chip scheduling of weighted sets of cubes

In order to fulfill the unique requirements for supporting stimuli generation with weighted distributions, a two-level address scheduling logic is designed based on the architecture shown in Figure 5.6. This is achieved by both arranging time slices among multiple sets of cubes and scheduling cubes in each set.

According to the specification for customized distribution in the up-to-date SystemVerilog standard (IEE, 2013a), the scheduling process should balance the total numbers of cycles for each set among the cube sets so as to be proportional to their weights during generation. Taking the constraint shown in Code 2.2 as an example, the two cube sets shown in Code 5.3 are interleaved during on-chip generation. For example, within a running period of 1,000 cycles, the total cycles for the two sets according to Code 2.2(a) and (b) should be 600 and 400 respectively, so as to comply with the weight ratio of 3:2. Note also, the stimuli generated according to each cube set should remain uniformly distributed. Therefore, the new addressing logic for scheduling multiple sets of cubes is designed based on the hardware framework which supports uniformly distributed stimuli generation, e.g., the structure elaborated in Section 5.5. Based on this framework, the addressing logic implements the round-robin strategy at the level of cube sets.

Figure 5.8 shows an example of the timeline for scheduling two cube sets, one of which has the weight of 3 and the other has 2. At the level of scheduling cube sets, the addressing logic periodically activates each set for a number of cycles equal to its weight one after another. When triggered to be active, each cube set can use its cubes independently. As shown in Figure 5.8, it adapts the strategy of round-robin to select between $\{a_i\}$ and $\{b_j\}$; meanwhile, it interleaves cubes within the time slices of each

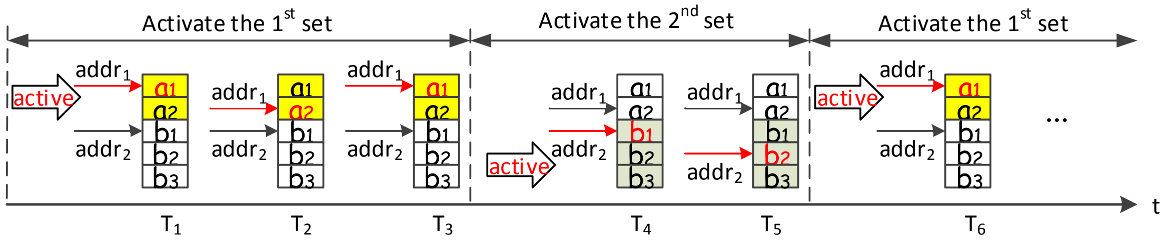


Figure 5.8: The timeline for scheduling two sets of cubes according to the weights of 3 for the set $\{a_i\}$ and 2 for the set $\{b_j\}$.

set independently. Considering the case in which all the cubes in a set are expired without additional updated cubes, the set of cubes are reset for scheduling again. Consequently, the two-level scheduling method realizes the weighted distribution as 3:2 in any five consecutive cycles and stabilizes around this ratio in the longer run.

The up-to-date SystemVerilog standard allows fractions for weights. In order to facilitate scheduling cube sets, the weights in the same **dist** expression are unified (simultaneously multiplied by a number) to positive integers, so that greatest common divisor of them is 1. For instance, the weights of 2/3, 1/4 and 1/6 in a **dist** expression are unified to 8, 3 and 2.

Regarding the volatility of the distribution during generation, given N cube sets ($N \geq 2$), each of which has a positive integer weight $w_i (1 \leq i \leq N)$, the addressing logic goes through all the sets once in each period of T cycles ($T = \sum_{i=1}^{N} w_k$). The expected ratio of the running time allocated to the i-th set is $r_i = w_i/T$, while the actual ratio for on-chip generation for t ($t \geq T$) cycles is $\bar{r}_i = t_i/t$ (t_i is the actual cycles consumed by the i-th set). Assuming that t includes τ ($\tau \geq 0$) periods of T, then $t = \tau T + \Delta t$ ($0 \leq \Delta t < T$) and $t_i = \tau w_i + \Delta t_i$ ($0 \leq \Delta t_i \leq \Delta t$). Considering the lower limit of $\bar{r}_i$ in the case of $\Delta t_i = 0$ (i.e., no active cycles are assigned to the k-th

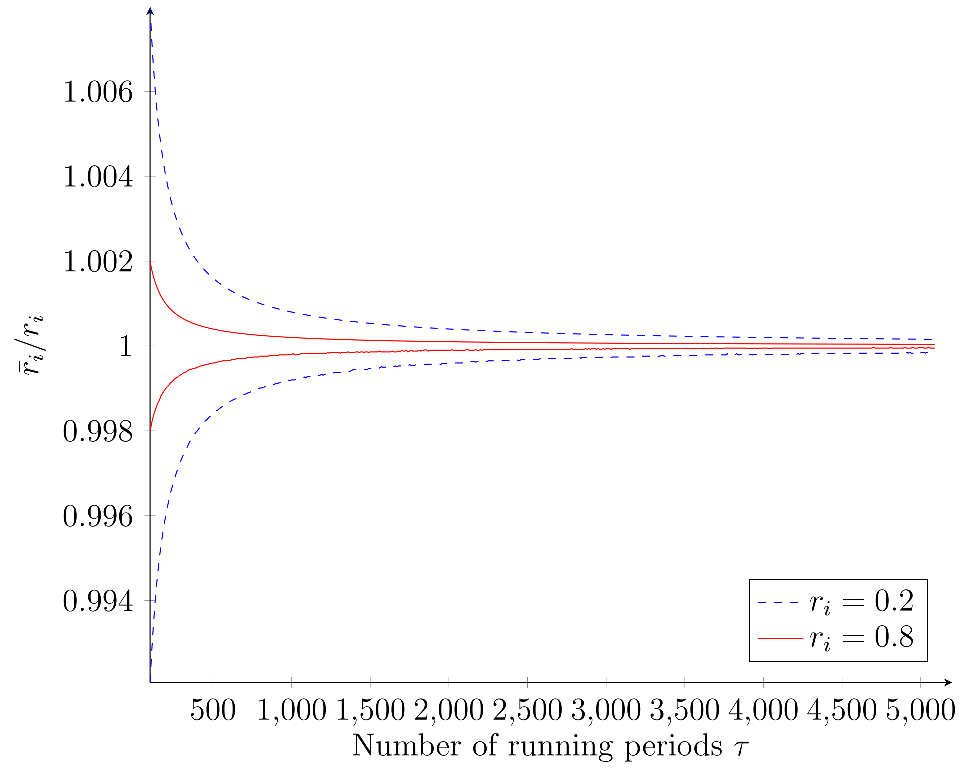


Figure 5.9: The actual distribution converges to the expected distribution.

set in Δt) and the upper limit if $t_i = \Delta t$, it can be deduced:

$$\frac{\tau w_i}{\tau T + T - w_i} \le \frac{\tau w_i}{\tau T + \Delta t} \le \bar{r}_i = \frac{\tau w_i + \Delta t_i}{\tau T + \Delta t} \le \frac{\tau w_i + w_i}{\tau T + w_i} \tag{5.2}$$

Hence the bounds for the ratio between r_i and $\bar{r}_i$ can be computed by Equation (5.3):

$$1 - \frac{1 - r_i}{\tau + 1 - r_i} \le \frac{\bar{r}_i}{r_i} \le 1 + \frac{1 - r_i}{\tau + r_i} \tag{5.3}$$

When generating the stimuli on-chip for long periods, the rate $\bar{r}_i/r_i$ is monotone convergent to 1. Figure 5.9 illustrates the curves for the upper limit and the lower limit according to Equation (5.3), which indicates that the actual distribution becomes close to the expected distribution when the on-chip generation iterates a few times through each cube set.

5.7 Experimental results

The solution of supporting constrained cyclic-random generation (introduced in Section 5.4) and the two extensions (introduced in Section 5.5 and Section 5.6) are evaluated in this section. In particular, the improvement, as well as the extra cost, for cube preparation and on-chip generation are examined and are compared against the solutions without distribution control that were presented in Chapter 4.

5.7.1 The evaluation of the algorithm for cube rectification

First the runtimes needed to generate non-overlapped cubes using Code 5.1 are evaluated. The experiments use the test cases with different types of constraints from Section 4.4. The results are shown in Table 5.3. Cases #1-#4 include solving inequalities based on integer linear programming constraints (shown in the second column), which typically specify numerical relationship between variables by a series of arithmetic, relational expressions. Cases #5 and #6 are designed to generate packet heads based on the formats of PCIe 3.0 TLP and H.264 RTP respectively. The third column gives the stimulus length and the fourth column shows the dimension of the constrained space from which valid patterns are sampled. The last column gives the runtimes, which clearly indicate the practical feasibility of the algorithm in Code 5.1.

The impact of the rectification process on the amount of final cubes is also evaluated, because it affects the size of the on-chip RAM resources. Figure 5.10 illustrates

Table 5.3: The test cases for assessing the runtime of Code 5.1.

Case No.	**Constraints**	**Stimulus length**	**Space size**	**Run time**
#1	$1000 \leq x + 2y \leq 8000$	24 bits	1.2×10^7	1.3s
#2	$y \geq 5x - 6000$, and (1)	24 bits	5.2×10^6	0.9s
#3	$x \geq 600$, and (1),(2)	24 bits	3.1×10^6	0.5s
#4	$y \geq 2000$, and (1),(2),(3)	24 bits	1.6×10^6	0.2s
#5	PCIe TLP	160 bits	8.7×10^{41}	3.5s
#6	H.264 RTP	168 bits	1.2×10^{50}	0.2s

the amount of the rectified (non-overlapped) cubes by the algorithm, compared with the amount of original cubes obtained by the solutions from Chapter 4. If the optional process of cube compaction is used, as it is the case for the solution in Section 5.4, the volume of data for the compact binary cubes (which are stored on-chip) is shown in Figure 5.11. These figures demonstrate that the algorithm in Code 5.1 can avoid pattern overlaps without causing a significant penalty in the volume of data that needs to be stored on-chip to configure the CRSGs in real-time. In fact, when cube compaction is used, the total volume of data can also be reduced when compared to the case when the cube overlaps are not removed (Figure 5.11).

5.7.2 The evaluation of the random-cyclic distribution

The quality of generated random-cyclic stimuli using the proposed methods is examined. It is compared with the results from the method shown in Section 4.3, which could also generate constrained-random stimuli as much as one stimulus per cycle but lacks the control on the distribution of stimuli. Note, both solutions from Section 5.4 and Section 5.5 support cyclic-random pattern generation.

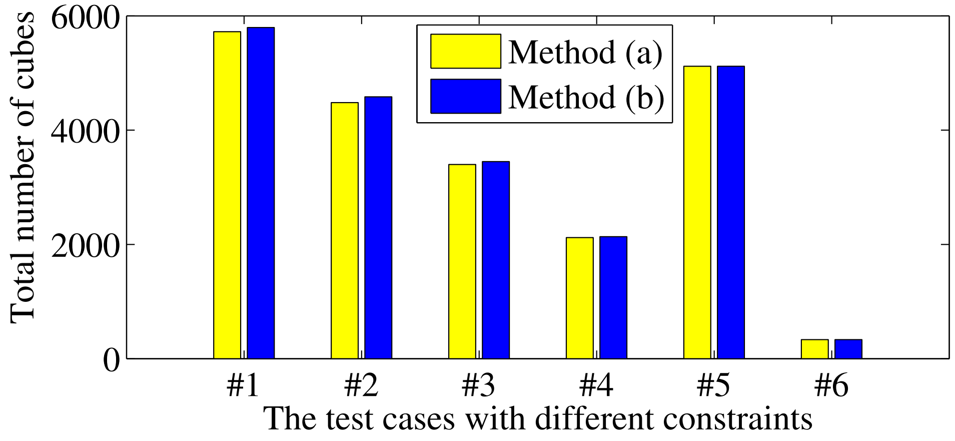


Figure 5.10: The number of cubes of each test case in the method (a) presented in Section 4.2 and the method (b) presented in Section 5.4.

Figure 5.12 illustrates the amount of unique stimuli within the first t generated stimuli for binary pairs $\langle x, y \rangle$ based on different constraints. Figure 5.12(a), using the constraint $x \geq y$, shows the speed-up for the exhaustive enumeration in a relatively small constrained sample space. In contrast, Figure 5.12(b), using the constraints #4 of Table 5.3, illustrates the result when the constrained sample space becomes larger. Because of the characteristics of the random-cyclic distribution, the trend of results is similar. That is, it generates t unique stimuli within t cycles in each permutation (a complete iteration over all the compliant values) regardless of the constraints.

5.7.3 The evaluation of area cost for random-cyclic stimuli generation

The comparison with generators without distribution control

The area cost (exclusive of RAM) of the proposed on-chip generator presented in Section 5.4 is estimated according to the maximum length of the supported stimuli

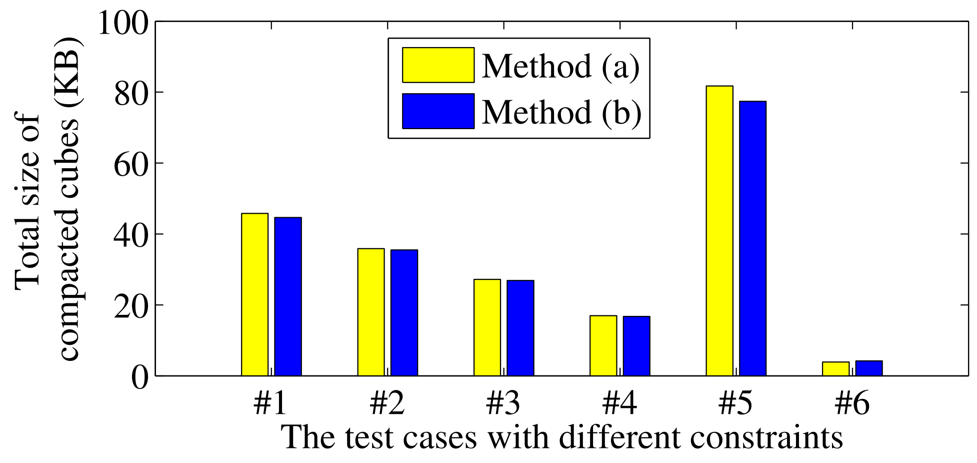
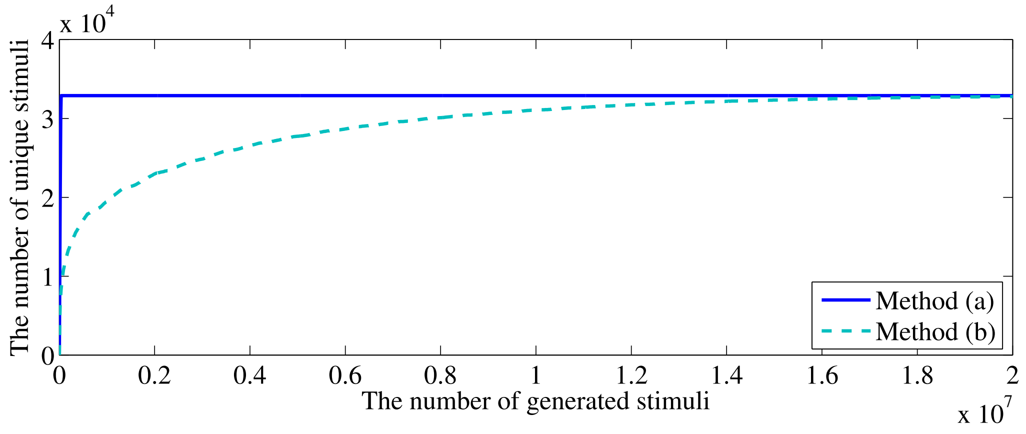


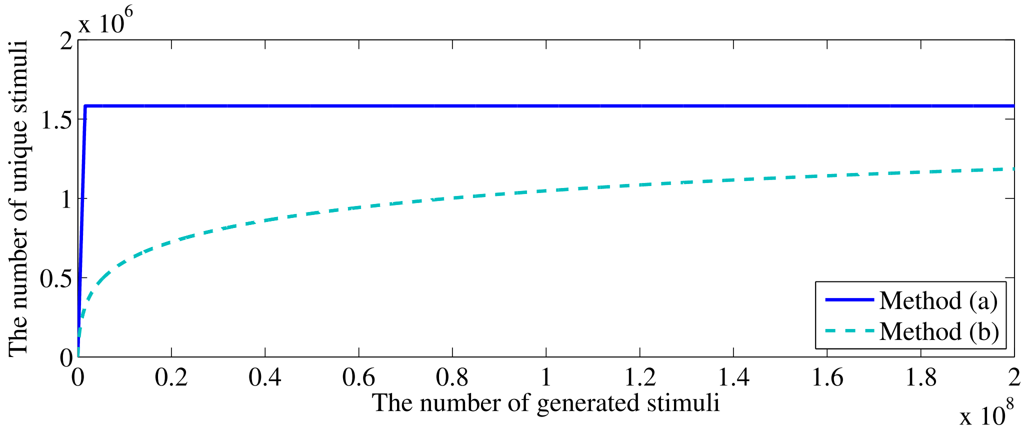
Figure 5.11: The volume of on-chip data (compacted cubes) for the method (a) presented in Section 4.2 and the method (b) presented in Section 5.4.

(m), as shown in Figure 5.13. The comparison is against the generators without distribution control, which were presented in the previous chapter. Note the results of the generator from Section 4.2 are plotted only for intuitive comparison, because its strategy for supporting longer stimuli ($m > 8$) is by consuming more cycles for decoding a cube, whereas the other methods quoted in Figure 5.13 enlarge the decoding hardware in order to process any cube within *one cycle*. As it can be noticed, the proposed generator only uses a low amount of extra hardware to implement the non-repetitive stimuli generation, when compared to the method from Section 4.3 that also processes one cube per clock cycle.

The critical path delay comparisons against the generator in Section 4.3 are given in Figure 5.14. The delay of the CRSG without pipeline stages is affected by the feedback paths from the decoding logic through vector assemblers to the stimulus output. Nonetheless, the data path can be partitioned using pipeline registers without any impact on the throughput. Hence the results for the 1-step pipelined generator are also given in the two figures, which confirm that the pipelining decreases the delay.



(a) The result for the constraint $x \geq y$ where x and y are 8-bit unsigned integers.



(b) The result for the constraints #4 shown in Table 5.3.

Figure 5.12: Assessing the repetition in the generated stimuli for the method (a) in Section 5.4 and the method (b) in Section 4.3.

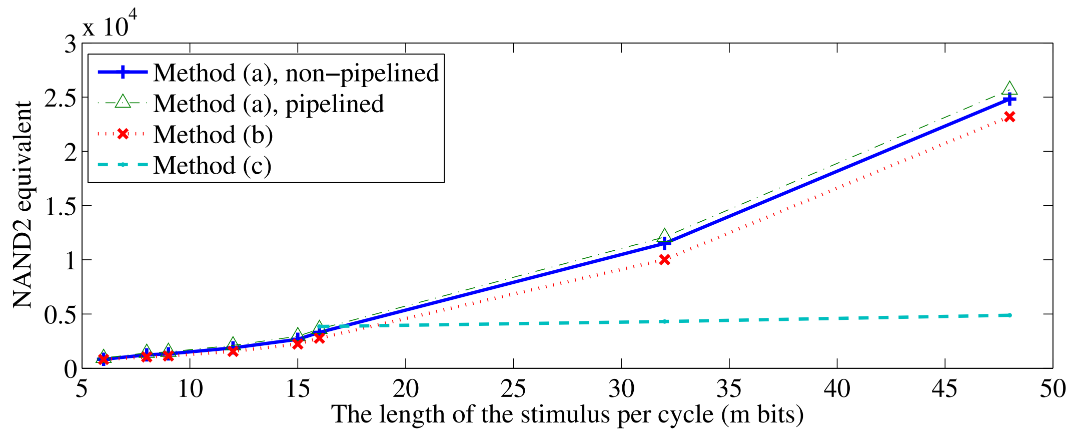


Figure 5.13: The hardware cost (exclusive of RAM) according to the length of supported stimuli for the method (a) (pipelined and non-pipelined) in Section 5.4, the method (b) in Section 4.3 and (c) in Section 4.2.

The comparison without the cube compaction

The on-chip generator, as presented in Section 5.5, is designed without cube decoding logic. Therefore it does not recognize compact binary cubes (in this case the area cost is reduced at the expense of more data to be stored on-chip). As shown in Figure 5.15, the area cost of this generator is against the area for the generator presented in Section 4.3. Note, the original generator presented in Section 4.3 also takes the area of on-chip decoding logic into account. In order to perform a fair comparison that could objectively show the extra cost for supporting on-chip distribution control, the results of the generator from Section 4.3 are estimated by excluding the area cost of the decoding logic. The method for random-cyclic distributed stimuli does require more hardware, and as the stimulus per cycle increases, the vector assembler dominates that area more than the other components (cube scheduler and the dynamic LFSR).

In terms of on-chip memory, the method presented in Section 5.5 that also supports cube interleaving uses an additional on-chip RAM for storing the context of each cube,

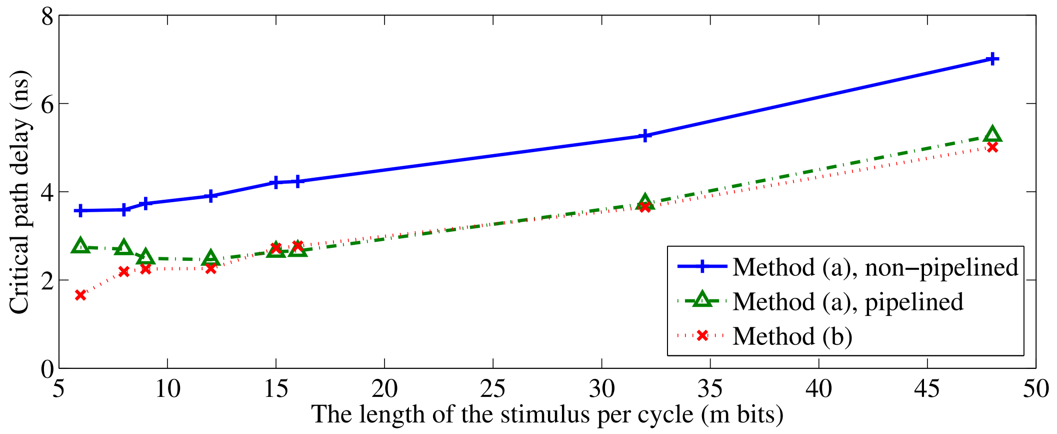


Figure 5.14: The critical path delay according to the length of supported stimuli for the method (a) (pipelined and non-pipelined) in Section 5.4 and the method (b) in Section 4.3. The data is collected from static timing analysis with a CMOS 90nm standard cell library.

as they get swapped during the scheduling process. Each context item contains two fields including k bits for the LFSR state and l bits to keep track how many patterns have been expanded from the corresponding cube (note, the maximum value is l will not exceed k). Also, assuming there are c cubes (and the byte alignment for each context item), the capacity of the context RAM is therefore equal to $c \times \lceil (k+l)/8 \rceil$.

5.7.4 The evaluation of the solution for supporting weighted distribution

The solution for generating stimuli according to the constraints with weighted distributions is evaluated, including the quality of the distribution during the dynamic generation process. The two-level addressing logic for scheduling multiple cube sets is implemented based on the generator for uniformly-distributed stimuli generation, as shown in Figure 5.6. The generator is configured to generate values according to

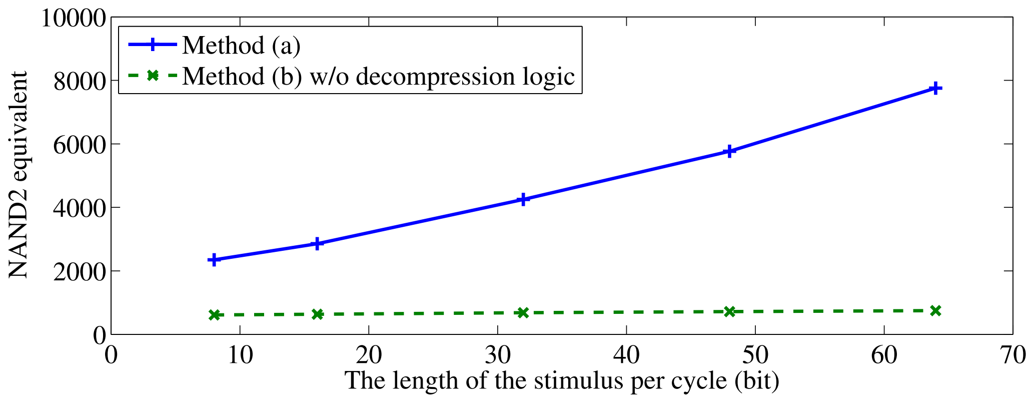


Figure 5.15: The hardware cost (exclusive of RAMs) of the generator from the method (a) in Section 5.5 and the generator without decoding logic from the method (b) in Section 4.3 according to the length of stimulus per cycle.

the weighted distributions of each range. Figure 5.16 illustrates the ratio of values that belong to the two ranges according to the constraint from Code 2.2. It shows that actual ratio of values in each range converges at the expected weight (0.6 and 0.4 respectively).

Figure 5.17 illustrates the area results by varying the number of supported cube sets. Area increases linearly, because the two-level addressing logic needs one more set of addressing registers in order to schedule an additional cube set.

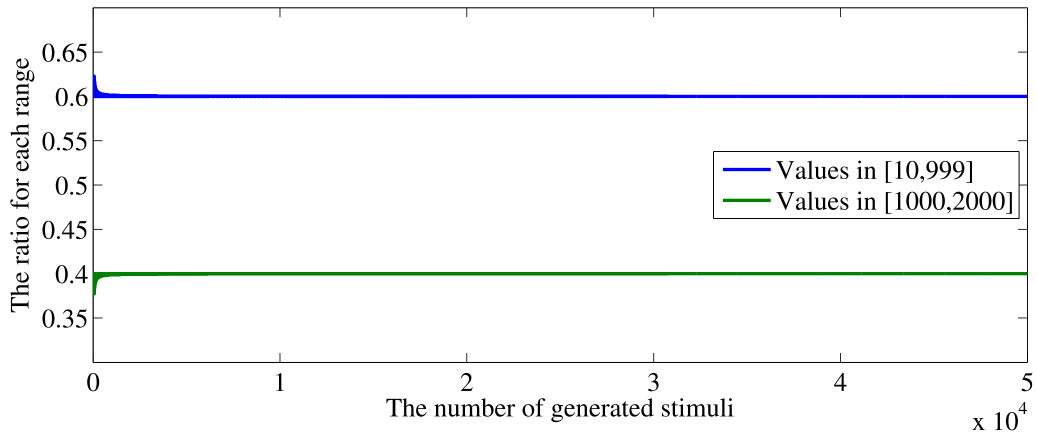


Figure 5.16: The ratio of values according to the constraint in Code 2.2 during on-chip generation.

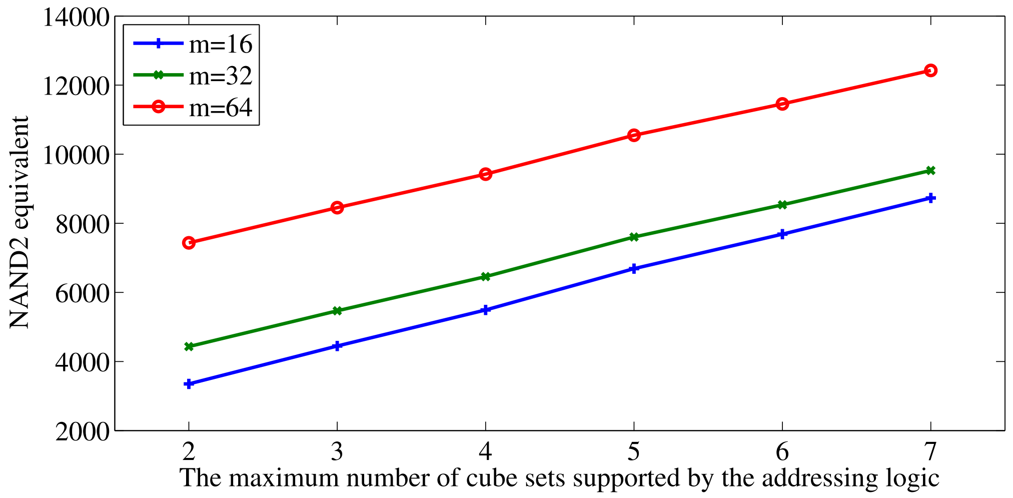


Figure 5.17: The area cost of the generator according to the number of supported cube sets. (The length of stimuli is m bits.)

5.8 Summary

In this chapter, solutions for on-chip constrained-random stimuli generation with distribution control have been presented. Special emphasis has been placed on the random-cyclic feature from SystemVerilog. A new algorithm, which re-shuffles the cubes to avoid distinct cubes to produce the same functionally-compliant patterns, has been presented. Using two new circuit blocks (dynamic LFSR and vector assembler) can avoid repetition of patterns causes by the same cube, thus ensuring that random-cyclic patterns (which are uniformly distributed) can be applied in real-time. Extra features have been added to avoid consecutive patterns to be constrained by the same cube, as well as to support weighted distributions during on-chip stimuli generation.

Chapter 6

Conclusion

Post-silicon validation is the critical step in the implementation flow of integrated circuits for exposing the subtle design errors that have escaped to the silicon prototypes. Its effectiveness is conditioned by in-system application of a large volume of functionally-compliant stimuli for extensive periods of time that can range from seconds to hours or even days. This is achieved by expanding on-the-fly randomized stimuli, which are subjected to user-programmable constraints, and monitoring the behaviour of the design under validation to determine if any design properties have been violated. This dissertation has focused on the controllability aspects of post-silicon validation and it has studied how to efficiently and cost-effectively apply large volumes of constrained-random stimuli in-system. The proposed solutions are programmable, thus enabling the users to update the constraints on the randomized stimuli at validation time.

In this chapter the main contributions of the work from this dissertation are summarized, followed by the suggestions for future work.

6.1 Summary of the contributions

In order to enable the reuse of the verification content from the pre-silicon stage in a post-silicon validation environment, several systematic methods based on content preparation and in-system configuration of on-chip signal generation circuitry have been proposed.

Central to the proposed methods is a cube-based representation of constraints expressed in a design and verification language, such as SystemVerilog. These cubes are used as masks that force pseudo-random sequences to map onto stimuli that are consistent with the constraints. The volume of data to be downloaded in-system could be further reduced by compacting cubes at design time and expanding them at validation time.

A method for designing programmable constrained-random signal generators has been presented. These generators can be placed on-chip and are parameterizable at design time and can be used to generate randomized stimuli in-system based on the configuration data derived from user-defined constraints. Both logic and sequential constraints can be supported and stimuli can be generated as fast as every clock cycle.

In order to support the control of the distribution of the constrained-random stimuli, a novel method for preparing cubes, as well as new circuit blocks for on-chip stimuli generation, have been developed. This method enables the generation of uniformly distributed, more specifically random-cyclic, stimuli within the user-constrained space. The results show that the process of cube preparation is fast and the area of the on-chip hardware is scalable. This method has been further expanded to support constraints that contain weighted distributions.

The features of the proposed solutions are summarized in Table 6.1.

Table 6.1: The summary of the proposed solutions for on-chip constrained-random stimuli generation.

Requirements	Functional constraints		Control of distribution		
Solutions for	Logic constraints	Sequential constraints	Uniform distribution	Interleaving cubes	Weighted distributions
Presented in	Section 4.2	Section 4.3	Section 5.4	Section 5.5	Section 5.6
Features[1]	Logic constraints	Sequential constraints	Uniformly/Random-cyclic distributed variables		Constraints with weighted distributions
See to the part of SystemVerilog	Functional operators, e.g., $+$, $>$, **if-else**; **function**[2]	**task**[2], **rand-sequence**	**rand/randc**		**dist**, **sovle-before**[3]

[1] The ability to change (reprogram) the constraints, as done during pre-silicon verification in SystemVerilog, is natively supported via dynamically reloading cubes to the on-chip generator.
[2] Restrictions apply according to the SystemVerilog standard (IEE, 2013a).
[3] Only for the constraints with **solve-before** that can be rewritten to constraints with value ranges with weighted distributions.

6.2 Suggestions for future work

The contributions from this dissertation have provided a step forward to bridge the gap between pre-silicon verification and post-silicon validation. To further improve the productivity, more tasks during the post-silicon validation stage will require systematic and automated methods.

Unlike during pre-silicon verification, where each block can have its own signal generator, in a post-silicon validation environment it is important to be conscious of the amount of hardware resources that are placed on-chip. Considering that modern system-on-a-chip devices contain tens to even hundreds of cores, it is worth investigating how to share the on-chip constrained-random signal generators across multiple cores. What and when can be shared is influenced by the parameters of the inputs for each core (e.g., bitwidth or throughput), as well as by the system interactions between cores (i.e., which blocks must be validated concurrently). Regarding the on-chip area cost, the interconnection between the shared CRSG and multiple blocks that are validated will not be negligible, and therefore it must be taken into consideration during place and route. If the wire delays become excessive, pipelining might be necessary to ensure that functional clock speeds will be satisfied. Performing a system-level analysis to decide where to place the signal generators, and how to share them across multiple validation sessions, is a topic worth further consideration.

The systematic way of offering in-system programmability remains an open research top which is worthy of further investigation. The methods for cube manipulation in the dissertation integrate a series of customized algorithms, as well as third-party tools, e.g., two-level minimization tool, hardware synthesis software and a BDD package, in order to achieve the goal of cube preparation. All these methods are practically usable. Nevertheless, the performance can be improved. For instance, if cube overlapping can be considered and solved at the beginning of cube generation process, the cube rectification process could be eliminated.

While the work from this dissertation has focused exclusively on the controllability aspects of post-silicon validation, a tighter interaction to the observability blocks is worth considering in the future. For example, as the validation process progresses, the on-chip monitors can provide "feedback" information to the on-chip constrained-random signal generators to drive the circuit state into corner cases that have not be sufficiently exercised. Also, if the system failures occur at validation time, by using the in-system programmability feature enabled by the work from this dissertation, the user can bias the stimuli in order to help narrow down the root cause of failure. Therefore, another interesting line for future research is to analyze the failing information off-chip and automatically suggest new sets of constraints (which can even be expressed directly as cubes) that can confirm or deny a potential cause of failure.

In summary, it is worth articulating that both pre-silicon verification and manufacturing test have matured into engineering disciplines that rely on systematic methods based on strong theoretical foundations. Post-silicon validation has traditionally relied on ad-hoc methods, with limited support from design automation tools. The lessons learned from the work presented in this dissertation add to the body of knowledge that can improve the productivity of the post-silicon validation tasks and enable more automated solutions in the foreseeable future.

Appendix A

The switching functions for the dynamic LFSR

The switch functions listed below are generated according to the characteristic polynomial table from (Živkovic, 1994). Only the on-set is shown in the table. For example, the entry

c_8: 8, 26, 27

indicates that $c_8 = 1 \iff \xi \in \{8, 26, 27\}$.

Table A.1: The on-set of switching functions for the 64-bit dynamic LFSR.

c_1 :	1, 2, 3, 4, 6, 7, 8, 13, 14, 15, 19, 22, 24, 26, 27, 30, 32, 34, 38, 42, 43, 44, 45, 46, 48, 50, 51, 53, 54, 56, 59, 61, 62, 63, 64
c_2 :	2, 5, 11, 1621, 35, 37, 40
c_3 :	3, 10, 12, 13, 16, 17, 20, 24, 25, 28, 31, 41, 45, 52, 64

c_4 :	4, 9, 13, 24, 39, 45, 64
c_5 :	5, 8, 16, 19, 23, 38, 43, 47
c_6 :	6, 8, 19, 38, 43
c_7 :	7, 12, 18, 26, 27, 57
c_8 :	8, 26, 27
c_9 :	9, 49
c_{10} :	10, 37
c_{11} :	11, 14, 36
c_{12} :	12, 14, 37
c_{13} :	13, 33
c_{14} :	14, 34
c_{15} :	15, 30, 34, 51, 53, 61
c_{16} :	16, 30, 51, 53, 61
c_{17} :	17
c_{18} :	18
c_{19} :	19, 40, 58
c_{20} :	20, 46
c_{21} :	21, 40, 46, 56, 59
c_{22} :	22, 42, 56, 59
c_{23} :	23, 42
c_{24} :	24, 55
c_{25} :	25
c_{26} :	26, 44, 50
c_{27} :	27, 32, 44, 48, 50
c_{28} :	28, 32, 48
c_{29} :	29
c_{30} :	30
c_{31} :	31
c_{32} :	32
c_{33} :	33
c_{34} :	34
c_{35} :	35
c_{36} :	36, 54
c_{37} :	37, 54
c_{38} :	38
c_{39} :	39
c_{40} :	40
c_{41} :	41
c_{42} :	42
c_{43} :	43
c_{44} :	44
c_{45} :	45
c_{46} :	46
c_{47} :	47
c_{48} :	48
c_{49} :	49
c_{50} :	50
c_{51} :	51
c_{52} :	52
c_{53} :	53
c_{54} :	54
c_{55} :	55
c_{56} :	56, 62
c_{57} :	57, 62
c_{58} :	58
c_{59} :	59
c_{60} :	60
c_{61} :	61
c_{62} :	62
c_{63} :	63
c_{64} :	64

Bibliography

(2006). IEEE standard for Verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–560.

(2008). IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70.

(2013a). IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315.

(2013b). IEEE standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pages 1–444.

Aagaard, M. D., Jones, R. B., and Serger, C.-J. H. (1999). Formal verification using parametric representations of boolean constraints. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 402–407. ACM.

Abramovici, M. (2008). In-system silicon validation and debug. *IEEE Transactions on Design & Test of Computers*, **25**(3), 216–223.

Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M., and Ziv, A. (2004). Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Transactions on Design & Test of Computers*, **21**(2), 84–93.

Adir, A., Copty, S., Landa, S., Nahir, A., Shurek, G., Ziv, A., Meissner, C., and Schumann, J. (2011). A unified methodology for pre-silicon verification and post-silicon validation. In *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

Ajanovic, J. (2009). PCI express 3.0 overview. In *Proc. Hot Chisp: A Symposium on High Performance Chips*.

Akers, S. B. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, **27**(6), 509–516.

Anis, E. and Nicolici, N. (2007). On using lossless compression of debug data in embedded logic analysis. In *Proc. IEEE International Test Conference (ITC)*, pages 1–10. Paper 18.3.

Bardell, P., McAnney, W., and Savir, J. (1987). *Built-in Test for VLSI: Pseudorandom Techniques*. Wiley-Interscience publication. Wiley.

Barnhart, C., Brunkhorst, V., Distler, F., Farnsworth, O., Keller, B., and Koenemann, B. (2001). OPMISR: the foundation for compressed ATPG vectors. In *Proc. IEEE International Test Conference (ITC)*, pages 748–757.

Bentley, B. (2001). Validating the Intel Pentium 4 microprocessor. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 244–248.

Bergeron, J., Cerny, E., Hunter, A., and Nightingale, A. (2006). *Verification Methodology Manual for SystemVerilog*. Springer US.

Boule, M., Chenard, J.-S., and Zilic, Z. (2007). Debug enhancements in assertion-checker generation. *IET Computers Digital Techniques*, **1**(6), 669–677.

Bryant, R. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, **C-35**(8), 677–691.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. (1990). Symbolic model checking: 10^{20} states and beyond. In *Proc. IEEE Fifth Annual Symposium on Logic in Computer Science*, pages 428–439.

Büttner, W. (1988). Unification in finite algebras is unitary(?). In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 368–377. Springer Berlin Heidelberg.

Cerny, E., Dudani, S., Havlicek, J., and Korchemny, D. (2014). *SVA: The Power of Assertions in SystemVerilog*. Springer International Publishing.

Chang, K.-H., Bertacco, V., and Markov, I. (2007). Simulation-based bug trace minimization with BMC-based refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(1), 152–165.

Cho, H., Hachtel, G., and Somenzi, F. (1993). Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **12**(7), 935–945.

Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proc. the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

Coudert, O. (1994). Two-level logic minimization: An overview. *Integr. VLSI J.*, **17**(2), 97–140.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, **5**(7), 394–397.

Fallah, F., Devadas, S., and Keutzer, K. (2001). Functional vector generation for HDL models using linear programming and Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20**(8), 994–1002.

Foster, H., Krolnik, A., and Lacey, D. (2012). *Assertion-Based Design*. Springer US.

Fujita, M., Tamiya, Y., Kukimoto, Y., and Chen, K.-C. (1991). Application of Boolean unification to combinational logic synthesis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 510–513.

Gharehbaghi, A. and Fujita, M. (2011). Formal verification guided automatic design error diagnosis and correction of complex processors. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 121–127.

Gherman, V., Wunderlich, H., Vranken, H., Hapke, F., Wittke, M., and Garbers, M. (2004). Efficient pattern mapping for deterministic logic BIST. In *Proc. IEEE International Test Conference (ITC)*, pages 48–56.

Goodenough, J. and Aitken, R. (2010). Post-silicon is too late avoiding the $50 million paperweight starts with validated designs. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 8–11.

Gopal, M. (1993). *Modern Control System Theory*. Wiley.

Hopkins, A. and McDonald-Maier, K. (2006). Debug support for complex systems on-chip: a review. *IEE Proc. Computers and Digital Techniques*, **153**(4), 197–207.

Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey. *The journal of logic programming*, **19**, 503–581.

Keshava, J., Hakim, N., and Prudvi, C. (2010). Post-silicon validation challenges: How EDA and academia can help. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 3–7.

Kilby, J. S. C. (2001). Turning potential into realities: The invention of the integrated circuit (Nobel lecture). *Chemphyschem: a European journal of chemical physics and physical chemistry*, **2**(8-9), 482–489.

Kinsman, A. B., Ko, H. F., and Nicolici, N. (2012). In-system constrained-random stimuli generation for post-silicon validation. In *Proc. IEEE International Test Conference (ITC)*, pages 1–10. Paper 3.3.

Kinsman, A. B., Ko, H. F., and Nicolici, N. (2013). Hardware-efficient on-chip generation of time-extensive constrained-random sequences for in-system validation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. Paper 39.6.

Kitchen, N. and Kuehlmann, A. (2007). Stimulus generation for constrained random simulation. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 258–265.

Ko, H. and Nicolici, N. (2009). Resource-efficient programmable trigger units for post-silicon validation. In *Proc. IEEE European Test Symposium (ETS)*, pages 17–22.

Ko, H., Kinsman, A., and Nicolici, N. (2008). Distributed embedded logic analysis for post-silicon validation of SOCs. In *Proc. IEEE International Test Conference (ITC)*, pages 1–10.

Köenemann, B. (1991). LFSR-coded test patterns for scan designs. In *Proc. IEEE European Test Conference (ETC)*, pages 237–242.

Kukula, J. H. and Shiple, T. R. (2000). Building circuits from relations. In *Proc. International Conference on Computer Aided Verification*, CAV '00, pages 113–123, London, UK, UK. Springer-Verlag.

Lee, C. Y. (1959). Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, **38**(4), 985–999.

Marques Silva, J. and Sakallah, K. (1996). GRASP-a new search algorithm for satisfiability. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 220–227.

Martin, U. and Nipkow, T. (1989). Boolean unification - the story so far. *Journal of Symbolic Computation*, **7**(3-4), 275–293.

McGeer, P., Sanghavi, J., Brayton, R., and Sangiovanni-Vicentelli, A. (1993). Espresso-signature: A new exact minimizer for logic functions. *IEEE Transactions on VLSI Systems*, **1**(4), 432–440.

Mentor Graphics, I. (2015). ModelSim ASIC and FPGA design.

Minato, S. (1996). *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer international series in engineering and computer science: VLSI, computer architecture, and digital signal processing. Springer.

Mitra, S., Seshia, S., and Nicolici, N. (2010). Post-silicon validation opportunities, challenges and recent advances. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 12–17.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC)*, pages 530–535. ACM.

Nahir, A., Ziv, A., Abramovici, M., Camilleri, A., Galivanche, R., Bentley, B., Foster, H., Hu, A., Bertacco, V., and Kapoor, S. (2010). Bridging pre-silicon verification and post-silicon validation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 94–95.

Nicolici, N. (2012). On-chip stimuli generation for post-silicon validation. In *IEEE High Level Design Validation and Test Workshop (HLDVT)*, pages 108–109.

Park, S.-B., Hong, T., and Mitra, S. (2009). Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **28**(10), 1545–1558.

Rajski, J., Tyszer, J., Kassab, M., Mukherjee, N., Thompson, R., Tsai, K.-H., Hertwig, A., Tamarapalli, N., Mrugalski, G., Eide, G., and Qian, J. (2002). Embedded deterministic test for low cost manufacturing test. In *Proc. IEEE International Test Conference (ITC)*, pages 301–310.

Sadasivam, S., Alapati, S., and Mallikarjunan, V. (2012). Test generation approach for post-silicon validation of high end microprocessor. In *Euromicro Conference on Digital System Design (DSD)*, pages 830–836.

Schliebusch, O., Meyr, H., and Leupers, R. (2010). *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer Netherlands.

Sivaraman, M. and Strojwas, A. (2012). *A Unified Approach for Timing Verification and Delay Fault Testing*. Springer.

Smith, M. (1997). *Application Specific Integrated Circuits*. Addison-Wesley VLSI systems series. Addison-Wesley.

Somenzi, F. (1999). Binary decision diagrams. In *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press.

Somenzi, F. (2012). CUDD: CU decision diagram package.

Spear, C. and Tumbush, G. (2012). *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer.

Stroud, C. (2002). *A Designer's Guide to Built-in Self-Test*. Frontiers in Electronic Testing. Springer.

Synopsys, I. (2015). VCS - functional verification solution.

Synopsys, I. (2016). Design Compiler - RTL synthesis and test.

Tang, S. and Xu, Q. (2008). In-band cross-trigger event transmission for transaction-based debug. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 414–419.

Touba, N. and McCluskey, E. (2001). Bit-fixing in pseudorandom sequences for scan BIST. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20**(4), 545–555.

Vermeulen, B., Waayers, T., and Goel, S. (2002). Core-based scan architecture for silicon debug. In *Proc. IEEE International Test Conference (ITC)*, pages 638–647.

Živkovic, M. (1994). A table of primitive binary polynomials. *Mathematics of Computation*, **62**(205), 385–386.

Wagner, I. and Bertacco, V. (2010). *Post-Silicon and Runtime Verification for Modern Processors*. SpringerLink : Bücher. Springer US.

Wang, L.-T., Wu, C., and Wen, X. (2006). *VLSI Test Principles and Architectures: Design for Testability*. Systems on Silicon. Elsevier Science.

Wang, Y.-K., Even, R., Kristensen, T., Tandberg, and Jesup, R. (2011). RTP payload format for H.264 video. RFC 6184.

Welp, T., Kitchen, N., and Kuehlmann, A. (2012). Hardware acceleration for constraint solving for random simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **31**(5), 779–789.

Weste, N. and Harris, D. (2011). *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley.

Wiemann, A. (2007). *Standardized Functional Verification*. Springer.

Wohl, P., Waicukauski, J., Patel, S., and Amin, M. (2003). X-tolerant compression and application of scan-atpg patterns in a BIST architecture. In *Proc. IEEE International Test Conference (ITC)*, volume 1, pages 727–736.

Wolf, W. (2004). *FPGA-Based System Design*. Pearson Education.

Wu, Y., Thomson, S., Mutcher, D., and Hall, E. (2011). Built-in functional tests for silicon validation and system integration of Telecom SoC designs. *IEEE Transactions on VLSI Systems*, **19**(4), 629–637.

Yuan, J., Aziz, A., Pixley, C., and Albin, K. (2004). Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **23**(3), 412–420.

Yuan, J., Pixley, C., and Aziz, A. (2010). *Constraint-Based Verification*. Springer US.

Index