UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PABLO RAFAEL BODMANN

Test Pattern Generator for Sequential Cells

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. Renato Perez Ribas

Porto Alegre
January 2018

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof^{a}. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Prof^{a}. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Renato Ventura Bayan Henriques
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Computers are like Old Testament gods; lots of rules and no mercy."

— JOSEPH CAMPBELL

AGRADECIMENTOS

Primeiramente, gostaria de agradecer aos meus pais pelo apoio e suporte durante todos esses anos. Gostaria de agradecer ao Prof. Renato Perez Ribas por me proporcionar um primerio contato no mundo acadêmico e por me incentivar em publicar o trabalho feito durante a bolsa de iniciação científica. Gostaria de agradecer ao Prof. André Inácio Reis e aos integrantes do grupo Logics pot todo apoio dado.

ABSTRACT

The validation of standard cell libraries used on digital integrated circuit design is a crucial task. However, the validation of sequential logic gates is quite complex due to the inherent memory effect found in these devices. In this work, it is proposed a generic test pattern generator to be applied on the validation of sequential cells. This generator is expected to be independent of the cell under test behavior, to change only one input per step and to be cyclic. To solve the problem, it is necessary to model this problem as a graph and find an Euler cycle over it. In order to find a cycle it is proposed the use of a modified Depth-first search. First the generator is validated using behavioral description of several different sequential cells. Also, is is validated using several different topologies. It is also proposed and analyzed the possibility of implementation on hardware.

Keywords: Sequential Cell. Testing. Digital Circuit. Standard Cell. Logic Gate.

Gerador de Padrões de Teste para Células Sequenciais

RESUMO

A validação de bibliotecas de Standard Cell usadas no design de circuitos integrados é uma tarefa crucial. No entanto, a validação dos portas seqüenciais é bastante complexa devido ao efeito de memória presente nestes dispositivos. Neste trabalho, propõe-se um gerador de padrões de teste genérico a ser aplicado na validação de células seqüenciais. Espera-se que este gerador seja independente do comportamento da célula sob teste, para alterar apenas uma entrada por etapa e seja cíclico. Para resolver o problema, é necessário modela-lo como um grafo e encontrar um ciclo de Euler sobre ele. Para encontrar um ciclo, propõe-se o uso de uma pesquisa por profundidade modificada. Primeiro, o gerador é validado usando a descrição comportamental de várias células seqüenciais diferentes. Também é validado usando várias topologias diferentes. Também é proposto e analisado a possibilidade de implementação em hardware.

Palavras-chave: Células sequenciais. Validação. Circuitos Digitais. Standard Cell. Portas lógicas.

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application-specific integrated circuit
BFS	Breadth First Search
BILBO	Built-in Logic Block Observer
BIST	Built-in Self-Test
CUT	Cell Under Test
DFS	Depth First Search
FFD	Type D Flip-Flop
FFDR	Type D Flip-Flop with asynchronous Reset
FFDS	Type D Flip-Flop with asynchronous Set
FFDRS	Type D Flip-Flop with asynchronous Reset and Set
FFT	Type T Flip-Flop
FFTR	Type T Flip-Flop with asynchronous Reset
FFTS	Type T Flip-Flop with asynchronous Set
FFTRS	Type T Flip-Flop with asynchronous Reset and Set
FPGA	Field-programmable gate array
FSM	Finite State Machine
IC	Integrated Circuit
Lat	Latch
LatD	Type D Latch
LatDR	Type D Latch with asynchronous Reset
LatDS	Type D Latch with asynchronous Set
LatDRS	Type D Latch with asynchronous Reset and Set
NCL	Null Convention Logic
PWL	Piecewise Linear

ROM	Read-only Memory
XOR	Exclusive-Or

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The design of integrated circuits (IC) is a lengthy process and it is composed of many phases. During these phases, the project passes through different levels of abstraction, from a behavioral description to a netlist of gates. Before going to the following step it is necessary that the current is working as expected in the specification.

Since it is interesting to validate the project during each phase as fast and accurate as possible, a good testing methodology is necessary. There are several testing strategies, from the use of external test benches to incorporate them inside of the chip. One of these that incorporate additional circuitry for testing with the component, is the concept of Built-in Self-Test (BIST) (BUSHNELL; AGRAWAL, 2013). This strategy is widely used because it reduces test and maintenance cost.

Another concept used to decrease time-to-market and cost of a project is the Standard Cell methodology. This concept uses pre-validated blocks and small circuits (named as cells) that implement logic functions. These pre-designed cells are available in a library. A library consists not only of different cells but also cells that implement the same logical function but with different topology or different drive strength (RABAEY, 1996).

When using any library-based approach, all of its cells must be correct before applying in a design. Since a single library comprises a large number of cells and all of them must be validated, the testing strategy must be efficient in order to reduce time and costs. Another problem encountered when validating libraries is the different types of cells that may comprise the library. This increases the difficulty of validation.

A cell library is composed of many types of cell. There are combinational cells, sequential cells, tri-states, buffers, I/O pads, fillers etc. Combinational cells are cells that implement functions whose output depends only on the current inputs. Sequential cells are the ones which have a memory effect, *i.e.*, for a combination of inputs, the output can be both logical values depending on the previous input sequence. The tri-state is a type of cell which the output can have a third state called high impedance, which the output is disconnected from both the source and ground nodes. Buffers are cells which regenerate the input signal. The other types don't implement a logical function, instead, they are used for several functionalities, from filling gaps in the design, to ensure the correct electrical behavior of the component.

1.1 Sequential Cell Validation

As discussed above, combinational cells and sequential have distinct logical behavior. Combinational cells are cells which implement Boolean functions. This means that for an input combination the output will always be same for that particular combination. For example, Table 1.1 shows the truth table for the combinational function exclusive-or (XOR). As can be seen each combination of "A" and "B" appear only one time. This means that for a combination of inputs there is only one possible output value.

Table 1.1: Exclusive-or truth table

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

This is not true for a sequential cell. An input combination can have both output values. Table 1.2 shows the truth table for a sequential cell called type D latch. As can be seen, for several input combination there are both possible outputs. This is called the memory effect. Therefore, for an input combination the output value cannot be directly known. Another difference is that some sequential cells have temporal limitations when transitioning inputs, and they must be respected otherwise the cell may go to an inconsistent output.

Table 1.2: Type D Latch truth table

E	D	Q
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	1	1

Testing combinational cells can be easily done with the solution proposed in (RIBAS et al., 2011b). In that work it is proposed a testing setup in which there are two blocks. One block has the cells-under-test and in the second there is a second stage which recover the input combination in order to make the input equal to the output. For combinational cells, it is an effective solution, however it is not appropriate for validation of sequential cells, because for an input combination may have both output values. This make the recovery of the input value more complex. Another problem is that more than one input may transition, violating the hold and setup time present in some sequential cells.

When validating a cell there are two main types of validation: the functional validation and the temporal validation. In the first, the expected behavior from the CUT is validated ensuring its correct functioning. In the second type it is validated the delay, *i.e.*, the time required for an input transition to propagate to the output.

In the context of sequential cell testing, there are several proposals with different strategies. At (MAKAR; MCCLUSKEY, 1995), it is proposed the use of Boolean equation describing the gate and create a corresponding state table in order to calculate the minimum input sequence to cover all possible defects usually observed in such cells. Another approach evaluates latches and flip-flops using shift-register and counter circuits, respectively (RIBAS et al., 2011c). An approach based on finite state machine, for describing the sequential gate behavior, is found in (AVELAR; BUTZEN; RIBAS, 2015). At (RIBAS et al., 2011a), an oscillating ring made of D type latches and flip-flops with asynchronous set and reset, is used to test the logical behavior and the temporal behavior of such cells. The works previously cited are either not generic, *i.e.*, the method used depends on of the cell behavior or does not cover all the cell-under-test (CUT) behavior.

1.2 Motivation

Validation is not a trivial task. Even combinational cells represent a challenge to IC designers. For example, Table 1.3 shows a malfunctioning 2-input OR logic gate. When both inputs are on the "1" logical value, instead of output "1", the gate outputs the previous output. If an IC designer test only the input combination, in crescent order for example, the fault will not be detected. This is accentuated in sequential cells which have a natural memory effect, thus, covering all input combinations is not sufficient, it is important to cover the transitions as well.

Table 1.3: Malfunctioning OR gate

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	Q_{-1}

Moreover, when using of the Standard Cell methodology, it implies that the cells used are pre-validated (AGATSTEIN; MCFAUL; THEMINS, 1990). Since sequential cells, especially latches and flip-flops, are essential in synchronous and asynchronous IC design, their correct validation is a necessity. Since there is a huge number of cells in a library, an efficient method is required. Using a generic pattern generator, *i.e.*, its behavior is independent of the CUT, can help in the process since a lot of different cells can be validated at the same time. Another problem is that a logic function can be implemented with many different topologies with different electrical behavior (ALIOTO; CONSOLI; PALUMBO, June 2011).

1.3 Proposal

In this work, it is proposed a pattern generator for the validation of sequential cell. It is desired that this generator in independent of CUT behavior, instead the generator behavior will be dependent only its output number. Another requirement is that the current pattern yielded by the generator differentiate form the next only by one output, *i.e.*, if both pattern would be represented by a bit sequence, they would have a Hamming distance of one. An additional requirement is that the generator would be cyclic, *i.e.*, it should start and end in the same pattern.

1.4 Organization

This work is organized as follows, in Chapter 2, it is discussed some essential concepts used in this work. In Chapter 3, some previous works are discussed. In Chapter 4, the generator is proposed, modeled and some hardware implementation are analyzed. In Chapter 5, the coverage of a set of cells are analyzed and four different sequential cell topologies are validated. In Chapter 6, the conclusions are outlined.

2 PRELIMINARIES

This chapter presents some important concept for the understanding this work. First, the different types of sequential cells and their logic behavior. Then it will be explained the different concepts such as the steady states and dynamic states (expected and unexpected transitions). Three basic gates are taken into account to illustrate these situations: C-element (Müller cell), D-type latch and D-type flip-flop, both with asynchronous reset signal.

2.1 Types of sequential logic circuits

There several types of sequential gates. Some are level sensitive, others are border sensitive and other have their behavior controlled by the current input combination. The most common of level-sensitive sequential gates,*i.e.*, their behavior is controlled by the logical value of the enable input, are latches (WESTE; HARRIS, 2010), for border-sensitive circuits, there are the flip-flops (WESTE; HARRIS, 2010). and for the third type there are the C-element and the NULL Convention Logic (NCL).

The C-element has its behavior when both inputs are equal the same logic value is presented at the output, and when the inputs are different, the gate output keeps its previous state (SPARS; FURBER, 2010). NCL is a method of representing an asynchronous gate. A gate using this method is represented as having k inputs, each with a weight. If all inputs are on the "0" logical value he output is "0". If the sum of the gates on the "1" logical value multiplied by their corresponding weights, the output will be on the "1" logical value, if not the gate holds its previous state(TRAN et al., 2017).

Latches and flip-flops are most commonly used in pipelines in synchronous architectures which have a global clock signal (HENNESSY; PATTERSON, 2011). However this circuits can have asynchronous Set and Reset signals, *i.e.*, these signals do not require other signals to force the output to a specific logical value. The C-element and NCL gates can be used in the data-path of asynchronous architectures which do not have a global clock signal (SPARS; FURBER, 2010).

Table 2.1: C-element steady states

A	B	Previous Q	expected Q
0	0	X	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	X	1

2.2 Steady States

The first aspect to analyzed is the Steady States coverage. The steady states are the possible combination of inputs and output values, *i.e.*, the truth table of the logical function which the circuit implements. Table 2.1 shows the steady states of the C-element, Table 2.2 presents the steady states of the D-type latch with asynchronous reset signal, and Table 2.3 shows the steady states of the D-type flip-flop with asynchronous reset.

Table 2.2: D-type latch with asynchronous reset steady states

R	D	E	Q
0	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	1	0	1
0	1	1	1
1	X	X	0

Table 2.3: D-type Flip-Flop with asynchronous reset steady states

R	D	CLK	Q
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	X	X	0

Since they are sequential circuits, they posses a natural memory effect. This reflects int the steady states as duplication of some input combinations. In the flip-flop, this effect is more significant. The reason for that is the fact that the latch is a level sensitive gate whereas the flip-flop is a border sensitive circuit. Therefore, the flip-flop presents

Table 2.4: C-element expected transitions

A	B	Previous Q	expected Q
1	↑	0	↑
↑	1	0	↑
↓	0	1	↓
0	↓	1	↓

more states to be covered. In order to cover these states, it is necessary to pass through a specific transition. The states with reset with the "1" logic value were omitted because the output is stuck at the 0 logical value. The C-element has an interesting behavior compared with latch and flip-flop gates. Instead of having the memory behavior controlled by a single input, in this case, it is controlled by both of them.

2.3 Dynamic States

2.3.1 Expected Transitions

Another important aspect of test coverage is the expected transitions, *i.e.*, when an input changes there is a transition at the output. As discussed before some sequential cells are border-sensitive therefore it is important to pass through all of them.

Table 2.4 shows the expected transitions of the C-element gate, Table 2.5 presents the expected transitions of the D-type latch, and Table 2.6 shows the expected transitions of the D-type flip-flop.

Table 2.5: D-type latch with asynchronous reset expected transitions

R	D	E	Previous Q	Q
0	0	↑	1	↓
0	↑	1	0	↑
0	↓	1	1	↓
↑	1	1	1	↓
↑	1	0	1	↓
↓	1	1	0	↑

Table 2.6: D-type flip-flop with asynchronous reset expected transitions

R	D	CLK	Previous Q	Q
0	1	↑	0	↑
0	0	↑	1	↓
↑	X	X	1	↓

Again, the difference among these circuits can be observed. The transition in the data input is only propagated to the output when the enable input is high or when the enable input rises the data input and this one differs from the current state of the latch. This propagation only occurs in the flip-flop when the clock signal rises and the data input differs from the current state of the flip-flop. The propagation of transitions in the C-element occurs when one input transitions to a value equal to the other, and this new input value differs from the current C-element state.

2.3.2 Non-Expected Transitions

Together with expected transitions are the non-expected transitions,*i.e.*, the transitions that occur when the input is transitioned but the output must stay stable. It is important to test the non-expected transitions, in order to evaluate the memory effect of the circuit. Table 2.7 shows the non-expected transitions for C-element, Table 2.8 presents the non-expected transitions for the latch, and Table 2.9 shows the ones for the flip-flop.

Table 2.7: C-element non-expected transitions

A	B	Previous Q	expected Q
↑,↓	0	0	0
↑,↓	1	1	1
0	↑,↓	0	0
1	↑,↓	1	1

Table 2.8: D-type latch with asynchronous reset non-expected transitions

R	D	E	Previous Q	expected Q
0	↑,↓	0	X	X
1	↑,↓	X	X	X
1	X	↑,↓	X	X
↑,↓	X	X	0	0
0	0	↑,↓	0	0
0	1	↑,↓	1	1

Table 2.9: D-type flip-flop with asynchronous reset non-expected transitions

R	D	CLK	Previous Q	expected Q
0	↑,↓	X	X	X
0	X	↓	X	X
↑,↓	X	X	0	0

The states show the memory effect of these circuits when a transition does not propagate to the output. These transitions must be covered in order to completely test whether the gate is holding the state and not transitioning.

3 REALTED WORKS

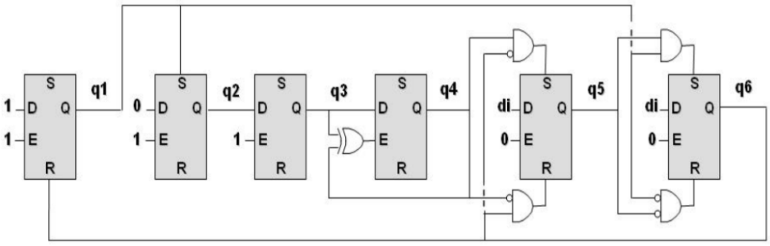
In this chapter, it is discussed some previous works which propose a solution for sequential cell testing. In each section, it is discussed the goal, the proposed design, the benefits and drawbacks.

3.1 Ring Oscillators for Functional and Delay Test

This work propose a design strategy using ring oscillators structure (ROS) for functional and delay test of latches and flip-flops (RIBAS et al., 2011a). The main goal is to validate the delay and power gate models applied and indirectly test the behavior of the cells. The maximum frequency of operation of the ROS is dictated by the time arches of the CUT.

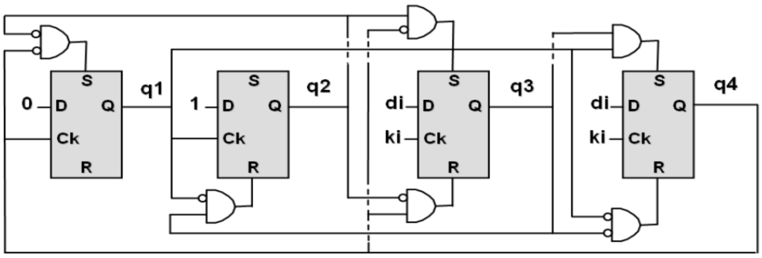
As concept of work, it is used as an example a type D latch and flip-flop with asynchronous set and reset. Figure 3.1 shows the setup for the latch and Figure 3.2 shows the setup for the flip-flop. In each stage a time arch from the inputs to the outputs is covered. The di and ki signals are controlled by an ROS made of a single flip-flop which receives its negated output.

Figure 3.1: Latch ROS described at (RIBAS et al., 2011a)



Source: (RIBAS et al., 2011a)

Figure 3.2: Flip-Flop ROS described at (RIBAS et al., 2011a)



Source: (RIBAS et al., 2011a)

The ROS is very suitable for self-checking and self-timed testing, because the operating frequency depends directly on the different timing arches. Besides, this structure can be easily converted to a synchronous execution mode by adding registers barrier and multiplexers. This test setup also covers all transitions of the cell using a small set of cells and a simple setup. Since the setup uses a ROS, it can be considered cyclic.

Despite having many benefits, these setups lacks some desired properties. It is shown only how to create a setup for latches and flip-flops, but not for other types of sequential cells, like C-elements that are asynchronous. Another problem is that the non-expected transitions are not covered and the steady states in which set and reset are on are not covered either. For the steady states of the flip-flop there are some other steady states that are not covered.

3.2 Checking Experiments to Test Latches

In this work, it is discussed the minimal checking experiment, *i.e.*, an input-output sequence that distinguishes a given cell, represented as a Finite State Machine, from another, for latches (MAKAR; MCCLUSKEY, 1995). The objective in this work is to show some sequence that are necessary in order to cover the most common fault found at these devices.

In order to determinate these minimum sequences, it is required that the CUT is represented as an equation. From it, it is extracted the possible steady states. Using a set of conditions it delimitates a checking sequence for the cell. For demonstrating the sequences ten different latches were used for demonstration.

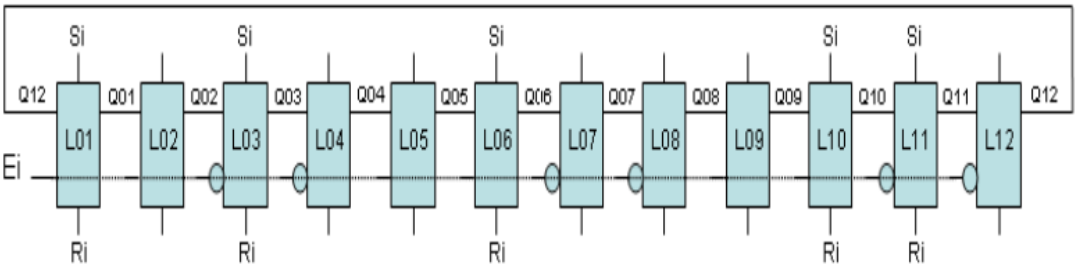
This strategy has a total coverage of states and detects any latch defect that do not increase the number of states. But, despite covering all steady states it does not cover all possible transitions possibly hiding errors caused by them. Another drawback is that it is not shown how to delimitate minimum test sequence for border sensitive sequential cells since they are difficult to model with a boolean expression. This strategy is also not cyclic.

3.3 Self-Checking Test Circuits for Latches and Flip-Flops

In this work, it is proposed two structures for the purpose of testing latches and for flip-flops (RIBAS et al., 2011c). The proposed setup use a self-timed and self-checking strategy in order to evaluate the CUTs. The proposed setup also is useful for delay test and power consumption analysis of these devices. Moreover, this setup can be used to evaluate the power supply variation and aging effect on such devices.

The test of latches is made with a shift-register with each element being the same as shown in Figure 3.3. However a couple of latches have their enable input negated. Also, some latches have their set and reset activated with the current state of the shift-register. The enable signal is also generated according to the general state of the register. This setup has a steady state coverage almost 100% but the state where both set and reset are on is not covered. Moreover, some possible and unexpected output transitions are not covered by this approach.

Figure 3.3: Shift-register proposed at (RIBAS et al., 2011c)

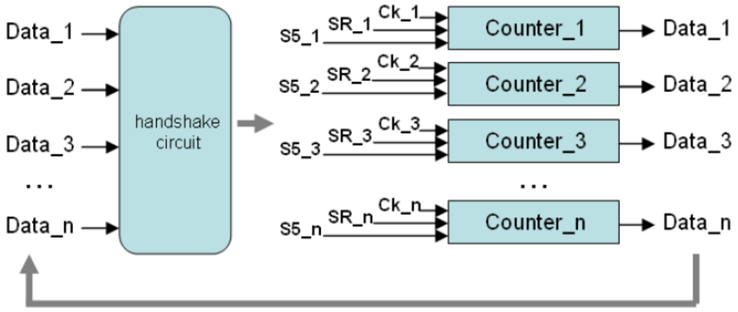


Source: (RIBAS et al., 2011c)

For testing flip-flop, the work proposed the use of several counters made of the CUT with a handshake circuit as shown in Figure 3.4. The clock, set and reset signals are generated by the handshake circuit which uses the overall state of the counter. This handshake circuit also halts if an error occurs. This setup covers all steady states and expected transitions but those where both set and reset are activated. The coverage of non-expected transitions are 50%.

These solutions, despite having high coverage and are cyclic, are not generic, *i.e.*, they are specific for latches and flip-flops only. It is not shown how to test other types of sequential cell.

Figure 3.4: Flip-flop testing setup proposed at (RIBAS et al., 2011c)



Source: (RIBAS et al., 2011c)

3.4 Automatic Circuit Generation for Sequential Logic Debug

This work propose an automated approach using finite-state-machines (FSM). This approach creates a test sequence and a testbench automatically in order to validate a sequential cell (AVELAR; BUTZEN; RIBAS, 2015).

Using a high level behavior description the possible steady states of the CUT are created and from them its possible transitions. The transitions are stored in a table and using a greedy algorithm to generate a sequence. From it a FSM is created using a hardware description language (HDL) code. The code is then synthesized along with the CUT.

This method has a total coverage of both steady and dynamic states. Since the CUT behavior is used, the method can be applied to any type of sequential cell. Moreover, it is topology independent if the cells have same behavior.

4 TEST PATTERN GENERATOR

In this work, it is proposed a generic pattern generator for testing sequential logic gates, *i.e.*, a generator independent from the CUT behavior. The generator provides signals in a cyclic sequence, *i.e.*, a sequence that starts and terminates at the same input vector. This is desired because the generator can be left running to stop only when an error occurs if implemented in circuit with Self-test methodology. It covers also all input states and signal transitions. The signal transitions occur by changing one bit per step. Such a characteristic is important because it avoids timing race conditions that can cause meta-stability and raise false-positive errors. Another reason is that it is easier to debug when an error occurs.

This work uses concepts similar to (MAKAR; MCCLUSKEY, 1995) and (AVELAR; BUTZEN; RIBAS, 2015) due to the use of a FSM strategy. The difference is that the proposed generator disregards the gates behavior and only focus the number of inputs that the CUT have. Another difference between these two previous works and the one proposed is that this generator is cyclic and can run multiple times without external signals if necessary.

4.1 Generator Modeling

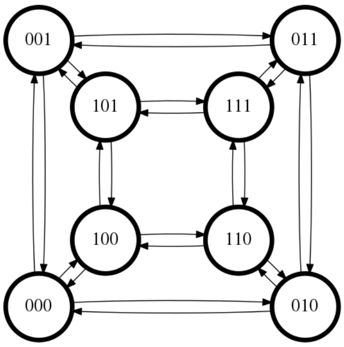
Since it is desired that the proposed generator is universal, its behavior must be independent from the CUT. Therefore, the cells memory effect is ignored and treated as a black box. One candidate for a testing sequence would be the Gray code (DORAN; SCIENCE, 2007) but it does not cover all possible states, as shown in Table 4.1. Thus, a better model is necessary in order to solve such a deficiency.

Table 4.1: 3-bit Gray code and its missing transitions.

Gray Code	Missing Transitions
000	$000 \rightarrow 010$ and $000 \rightarrow 100$
001	$001 \rightarrow 101$ and $001 \rightarrow 000$
011	$011 \rightarrow 001$ and $011 \rightarrow 111$
010	$010 \rightarrow 011$ and $010 \rightarrow 000$
110	$110 \rightarrow 010$ and $110 \rightarrow 100$
111	$111 \rightarrow 110$ and $111 \rightarrow 011$
101	$101 \rightarrow 111$ and $111 \rightarrow 001$
100	$100 \rightarrow 101$ and $100 \rightarrow 110$

Despite the Gray code not meeting the requirements to be used as a test pattern generator, a model can be built using the contents from Table 4.1. The possible inputs are represented as nodes in a graph and the edges the possible transitions. Since it is interesting to cover both rising transition, when a bit goes from logic value 0 to value 1, and falling transition, when a bit goes from logic value 1 to value 0, the graph must be bidirectional. Figure 4.1 shows the model for a 3-input cell generator.

Figure 4.1: 3-Dimensional Hypercube representing a 3 bit Gray Code



Source: The Author

Once having built the graph, it is only necessary to find out an Euler cycle, *i.e.*, a cycle that passes through all edges exactly once, beginning and ending at the same node. Due to the form of the graph, it has many different Euler cycles, therefore it is proposed a simpler solution.

A modified DFS (depth first search) is used in order to create the sequence. The algorithm only jumps to nodes with larger labels than the current one. DFS was chosen over BFS (breadth first search) because DFS gives the smallest vector sequence possible. For instance, if the algorithm is in the node "001" when using DFS, then the algorithm travels to the nodes that are closer to it (its sons). But using BFS, the algorithm needs to explore the other nodes at the same level on the other branches of the tree before going deeper. This causes a lot of unnecessary transitions. In order to keep 1-bit transition rule, the algorithm must travel through the tree and pass over already covered transitions.

4.2 Software Implentation

In order to demonstrate the generator functioning, a Python application was created. This application receives the desired pin number and outputs the resulting sequence. The graph is represented in a modified adjacency list. Instead of showing every edge in the graph, it is only shown the edges to the bigger neighbors. This can be calculated by verifying each bit from the value and if it is "0" it is flipped and saved as a next node in the current list. Figure 4.2 shows the pseudo-code for the creation of the table. Using as an example for a 3-bit generator, a table was created in order to illustrate. Table 4.2 shows the expected result for the given example. With the list built it is only necessary to make the DFS.

Figure 4.2: Pseudocode for creating Table of adjacency lists

```
**Input:** The number of inputs of the CUT
**procedure** $Create\_Table(N)$
    $adjacency\_table = [[] * 2^N]$
    **for** $i = 0; i < 2^N; i++$ **do**
        $mask = 1$
        **for** $j = 0; j < N; j++$ **do**
            **if** $temp \wedge mask == 0$ **then**
                $temp = i \oplus mask$
                $adjacency\_table[i].append(temp)$
            **end if**
            $mask \ll 1$
        **end for**
    **end for**
    **return** $adjacency\_table$
**end procedure**
```

Source: The author

Table 4.2: Table containing each node adjacency list representing the graph in Fig. 4.1

Nodes	000	001	010	011	100	101	110	111
Next Nodes	001	011	011	111	101	111	111	
	010	101	110		110			
	100							

After the table is created, the search through it can be made and the pattern sequence can be created. Using as an example a 2-input generator and a 3-input generator, as shown in Table 4.2, the algorithm will output the sequences shown in Table 4.3 and Table 4.4, respectively.

Table 4.3: 2-input sequence

2-input Sequence	00	01	11	01	00	10	11	10

Table 4.4: 3-input Sequence

3-input Sequence	000	001	011	111	011	001	101	111
	101	001	000	010	011	010	110	111
	110	010	000	100	101	100	110	100

Seeing the contents of Table 4.3 and Table 4.4, it can be seen that all transitions are covered. But this growth can be easily calculated. For N inputs, there will be 2^N possible combinations. Each combination will have N neighbors, because of the 1-bit transition rule. Therefore, in total, the sequence will have $N * 2^N$ steps in order to cover all transitions. Table 4.5 shows the length of the sequence with more inputs.

Table 4.5: Sequence growth

# of inputs	length of the sequence
2	8
3	24
4	64
5	160
6	384
7	896
8	2048
9	4608
10	10240

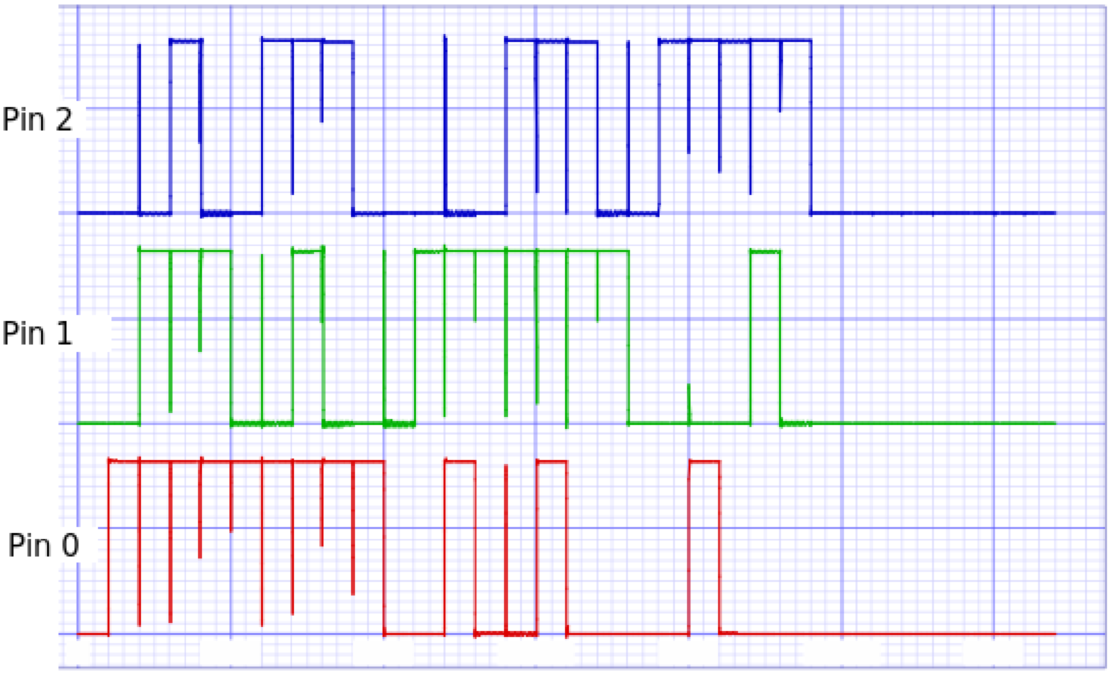
4.3 Hardware implementation

Since the proposed generator is independent of the CUT and it produces a cyclic sequence, it is possible to use it in a Self-test setup when validating in hardware. Therefore, it was investigated a chip size estimation.

4.3.1 ROM Solution

One proposed physical implementation is the use of a read-only memory (ROM). In order to test this approach, a script was made. This application receives as input the desired generator size and creates the sequence. After, it automatically creates a SPICE netlist, which implements a decoder and the ROM matrix. In each line of the matrix is an input vector. Since the generator grows 2^N, the resulting memory must be capable of mapping at least $N * 2^N$ addresses. In order to profit of the full capacity of the memory, the sequence can be parted in N banks of memory with a capacity for 2^N addresses but the simulated ROM has a single memory bank for simplicity and the memory lines, which were not used, were filled with the first state (0). Figure 4.3 shows a simulation on SPICE of a ROM design. One problem observed is the voltage spikes during each transition. This can be avoided using a register in the memory output.

Figure 4.3: Simulation of ROM implementing a 3-input generator



Source: The author

4.3.2 Boolean Network and LUT Based Solutions

Since the generator can be modeled as a list of input patterns, it is easy to create a behavioral description in a Hardware Description Language (HDL) such as Verilog. So a script was created that creates a sequence and put it in a Verilog module. The module receives as input as an index and outputs the corespondent vector. After that, using a commercial synthesis tool, the Verilog code was transform into a Boolean Network consisting of the following primitives: NAND, AND, NOR, OR, XOR, XNOR and NOT. Besides that it was synthesized to a FPGA and a 4-input look-up table (4-LUT) network was created. As expected, both implementations grow exponentially as the implementation using a ROM, since the number of patterns grows exponentially as well.

Figure 4.4: Growth behavior of the Boolean network per number of inputs

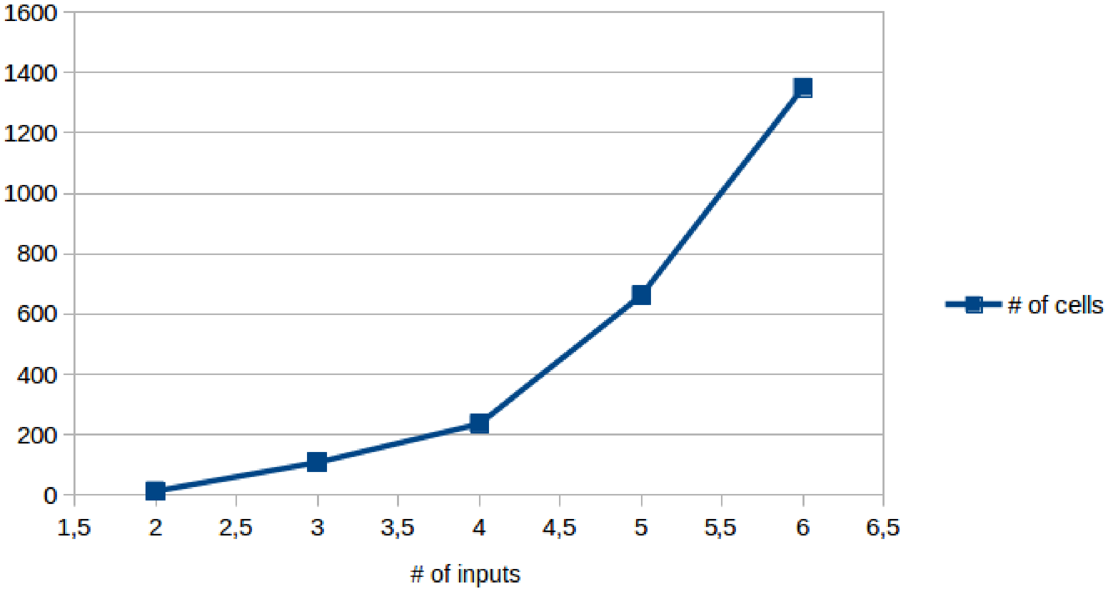
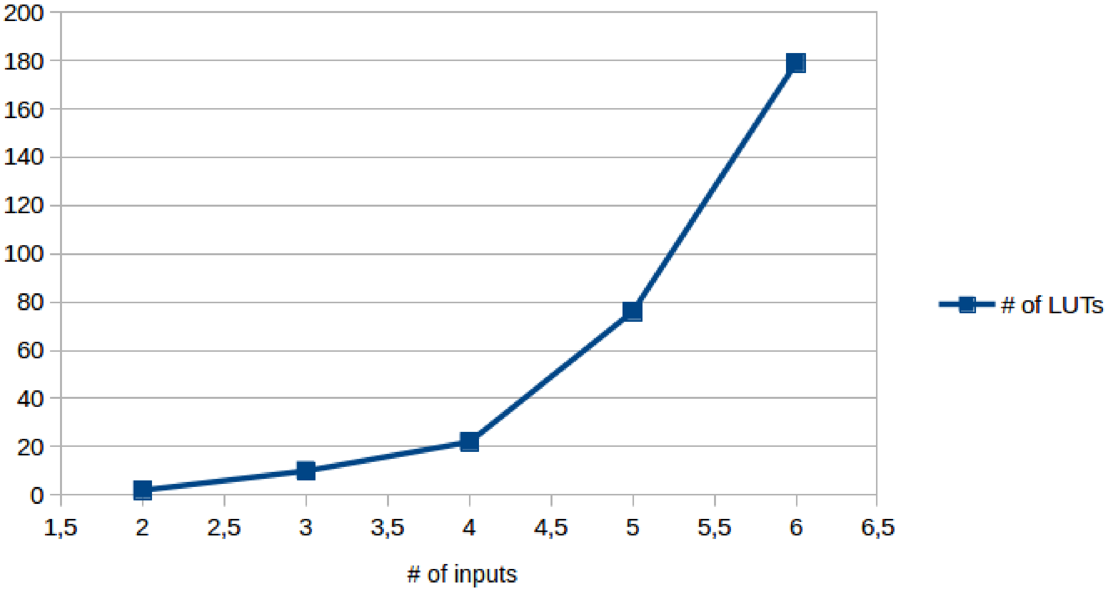


Figure 4.5: Growth behavior of the FPGA per number of inputs



5 VALIDATION OF SEQUENTIAL CELLS

In this chapter, it is discussed the effectiveness of the proposed generator in validating sequential cell. Several types of cells were used and their steady and dynamic coverage analyzed.

5.1 Model Validation

In order to do so, another application in Python was made, in which different types of cells have their behavior described. Then, they were stimulated with the generators sequence and the coverage of both states recorded. Besides the normal latches and flip-flops, several other types cells were used, and they are listed below with a brief description. The selected cells were stimulated with 3 iterations of the sequence in order to determinate the total coverage. It was assumed that all cells begin in the "0" state when possible.

BILBO Latch A latch used in a BILBO register which changes its behavior depending of the coombination of the inputs B1 and B2.

C element A sequential cell, which behavior depend on the combination of its inputs.

Dual edge FFD This type of FFD propagates the Data input to the output on both rising and falling edge.

Scan FFD and LatD This kind of FFD and LatD has a multiplexer in the D input and a selector signal.

FFT The T type flip-flop inverts the current output when the "T" input is on and the clock is rising.

Gated FFD This type of FFD has an additional a enable signal which enables the clock signal.

Two Port latch This kind of LatD has two inputs and two corresponding enable signals. If both signals are on an OR operation is made with both inputs.

Xor FF This kind of flip-flop has two "Data" input and a XOR operation is made between them.

5.1.1 Single instance validation

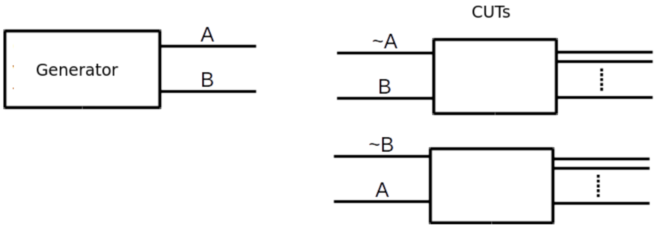
First, it was tested if using only 1 instance of the cell would be enough in order to cover all steady and dynamic states. In this first test the order of inputs used was the one described in the cells behavior. Table 5.3 shows the cells used for testing, their input ordering and both coverage for the cells used. For the Null Convention Logic (NCL) description, it is shown the weight for each input. Using only a single cell, only one CUT managed to get full coverage, the T flip-flop.

The reason for the other cells not being fully covered is the natural memory effect of these cells and the asynchronous signals that may interfere with the current state of the cell. The generator sequence despite covering all inputs transitions and possible states does not cover all steady and dynamic states of the cell.

5.1.2 Multiple instances validation

Seeing that the generator alone does not fully validate the cells, improvements must be made. It is interesting to maintain the sequence independent of the CUT behavior so the only possibility is to use multiple instances of the CUT but with some inputs permuted or negated, as shown in Figure 5.1.

Figure 5.1: Proposed improvement



Source: The author

Table 5.1: Cell with partial coverage

CUT	State not covered
FFD	[UP, 1, 1]
FFDR	[UP, 1, 0, 1]
FFTR	[UP, 1, 0, DOWN]
gated FFD	[1, UP, 1, 1]
gated FFDR	[1, UP, 1, 0, 1]

Using the improvement described, the selected CUTs were again stimulated with the generators sequence. Since some inputs are negated, the initial output was corrected to the appropriate value. All of them had total coverage for steady states but some did not have total coverage for dynamic states. Table 5.4 shows the number of instances used and Table 5.1 shows which cells did not have total coverage for both type of states and which states were not covered. It is interesting to notice that the number of instances approach the number used in (RIBAS et al., 2011c) but in this case the generator can be used for any type of cell. Even for complex cells represented in NCL the number is acceptable.

5.1.3 Continuous validation

As discussed before, the generator can be implemented in hardware. However, despite the generator producing a cyclic sequence, some CUTs change their state after the first cycle. This introduces a problem, when the generator return to the first state of the second cycle, the template will not match and a false positive error will be raised.

In order to avoid this problem, when validating on hardware, two solutions are proposed. One can use a previous validated cell in order to use as a template, or the first iteration is run without checking with the template. So, it is necessary to use only instances of the cell which has the same state after the first cycle. In order to investigate the coverage behavior when ignoring the first iteration, the sequence was run three times but the first iteration was ignored. Table 5.2 shows the states not covered for the cells which did not have total coverage and Table 5.5 shows the coverage and the number of cells used for each CUT.

It is interesting to notice that some cells had the coverage decreased, or some had the number of instances needed increased. This means that some Dynamic states are covered only if the first cycle.

Table 5.2: States not Covered when disregarding the first iteration

Cell	States not Covered
FFD	[UP, 0, 0], [UP, 1, 1]
FFDR	[UP, 1, 0, 1]
FFDS	[UP, 0, 0, 0]
FFTR	[UP, 1, 0, DOWN]
FFTS	[UP, 1, 0, UP]
gated FFD	[1, UP, 0, 0], [1, UP, 1, 1]
gated FFDR	[1, UP, 1, 0, 1]
gated FFDS	[1, UP, 0, 0, 0]

Table 5.3: Cells used with input ordering and coverage

CUT	Input Ordering	Steady Coverage	Dynamic Coverage
BILBO Latch	Enable, B1, B2, Qp, Zp	91.67%	67.08%
C element	NA	100,00%	66.66%
Dual Edge FFD	Clock, Data	87.5%	56.25%
Dual Edge FFDR	Clock, Data, Reset	83.33%	66.66%
Dual Edge FFDS	Clock, Data, Set	100.0%	69.44%
Dual Edge FFDRS	Clock, Data, Reset, set	100.0%	81.25%
FFD	Clock, Data	87.5%	56.25%
FFDR	Clock, Data, Reset	83.33%	66.66%
FFDS	Clock, Data, Set	91.67%	69.44%
FFDRS	Clock, Data, Reset, set	100.0%	81.25%
FFD scan	Clock, Scan Sel, Scan Data, Data	84.37%	50.78%
FFD scan R	Clock, Scan Sel, Scan Data, Data, Reset	75.0%	66.66%
FFD scan S	Clock, Scan Sel, Scan Data, Data, Set	97.91%	67.08%
FFD scan RS	Clock, Scan Sel, Scan Data, Data, Reset, Set	100.0%	80.0%
FFJK	K, J	81.25%	52.08%
FFT	Clock, T	100.0%	100.0%
FFTR	Clock, T, Reset	83.33%	66.66%
FFTS	Clock, T, Set	100.0%	75.0%
FFTRS	Clock, T, Reset, set	100.0%	83.75%
gated FFD	Enable, Clock, Data	81.25%	54.16%
gated FFDR	Enable, Clock, Data, Reset	75.0%	66.66%
gated FFDS	Enable, Clock, Data, Set	83.33%	68.75%
gated FFDRS	Enable, Clock, Data, Reset, Set	100.0%	80.0%
LatchD	Enable, Data	100.0%	75.0%
LatchDR	Enable, Data, Reset	90.0%	80.0%
LatchDS	Enable, Data, Set	100.0%	83.33%
LatchDRS	Enable, Data, Reset, Set	100.0%	90.27%
LatchD scan D	Enable, Scan Sel, Scan Data, Data	95.83%	67.70%
LatchD scan R	Enable, Scan Sel, Scan Data, Data, Reset	81.81%	72.72%
LatchD scan S	Enable, Scan Sel, Scan Data, Data, Set	100.0%	80.5%
LatchD scan RS	Enable, Scan Sel, Scan Data, Data, Reset, Set	100.0%	88.88%
NCL1	[1,2,3,6,12,18]	95.34%	75.14%
NCL2	[6,5,4,2,2,1]	89.21%	63.36%
NCL3	[4,3,1,1,1,1]	85.71%	61.53%
SR latch	R, S	100.0%	90.0%
Two Port Latch	Enable 1, Data 1, Enable 2, Data	100.0%	81.25%
Two Port Latch R	Enable 1, Data 1, Enable 2, Data, Reset	97.22%	88.88%
Two Port Latch S	Enable 1, Data 1, Enable 2, Data, Set	100.0%	89.44%
Two Port Latch RS	Enable 1, Data 1, Enable 2, Data, Reset, Set	100.0%	94.36%
XOR FF	Clock, Data 1, Data 2	87.5%	52.08%
XOR FF R	Clock, Data 1, Data 2, Reset	87.5%	66.66%
XOR FF S	Clock, Data 1, Data 2, Set	87.5%	67.70%
XOR FF RS	Clock, Data 1, Data 2, Reset	100.0%	80.0%
XOR Latch	Enable, Data 1, Data 2	100.0%	69.44%
XOR Latch R	Enable, Data 1, Data 2, Reset	95.0%	80.0%
XOR Latch S	Enable, Data 1, Data 2, Set	95.0%	81.25%
XOR Latch RS	Enable, Data 1, Data 2, Set, Reset	100.0%	88.88%

Table 5.4: Number of instance used

Cell	# of instances
BILBO Lat	5
C element	2
Dual Edge FFD	3
Dual Edge FFDR	4
Dual Edge FFDS	4
Dual Edge FFDRS	4
FFD	3
FFDR	6
FFDS	5
FFDRS	3
FFD scan	4
FFD scan R	12
FFD scan S	7
FFD scan RS	4
FFJK	3
FFT	1
FFTR	4
FFTS	4
FFTRS	2
gated FFD	2
gated FFDR	9
gated FFDS	8
gated FFDRS	3
LatD	2
LatDR	4
LatDS	3
LatDRS	3
LatD scan D	4
LatD scan R	7
LatD scan S	3
LatD scan RS	3
NCL1	30
NCL2	43
NCL3	39
SR Lat	2
Two Port Lat	4
Two Port Lat R	5
Two Port Lat S	6
Two Port Lat RS	3
XOR FF	3
XOR FF R	8
XOR FF S	8
XOR FF RS	4
XOR Lat	3
XOR Lat R	4
XOR Lat S	4
XOR Lat RS	3

Table 5.5: Coverage and number of cells used disregarding the first iteration

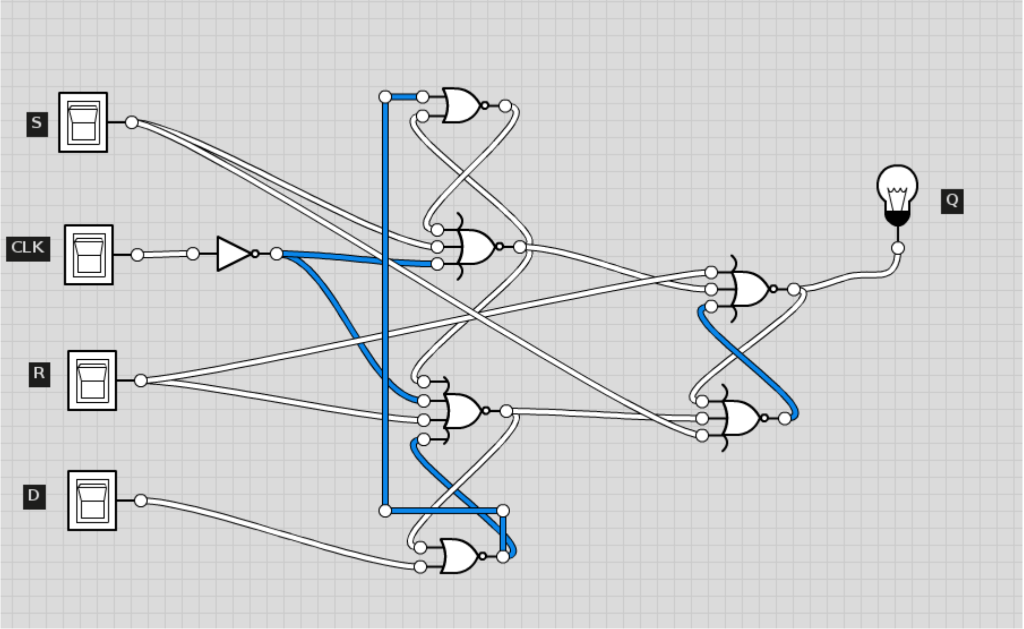
CUT	# of cells	Steady Coverage	Dynamic Coverage
BILBO Latch	5	100,00%	100,00%
C element	2	100,00%	100,00%
Dual Edge FFD	4	100,00%	100,00%
Dual Edge FFDR	4	100,00%	100,00%
Dual Edge FFDS	4	100,00%	100,00%
Dual Edge FFDRS	4	100,00%	100,00%
FFD	4	100,00%	87,50%
FFDR	6	100,00%	97,22%
FFDS	6	100,00%	97,22%
FFDRS	3	100,00%	100,00%
FFD scan	4	100,00%	100,00%
FFD scan R	12	100,00%	100,00%
FFD scan S	7	100,00%	100,00%
FFD scan RS	4	100,00%	100,00%
FFJK	5	100,00%	100,00%
FFT	1	100,00%	100,00%
FFTR	4	100,00%	97,22%
FFTS	4	100,00%	97,22%
FFTRS	2	100,00%	100,00%
gated FFD	4	100,00%	95,83%
gated FFDR	9	100,00%	98,95%
gated FFDS	10	100,00%	98,95%
gated FFDRS	3	100,00%	100,00%
LatchD	3	100,00%	100,00%
LatchDR	4	100,00%	100,00%
LatchDS	4	100,00%	100,00%
LatchDRS	3	100,00%	100,00%
LatchD scan D	4	100,00%	100,00%
LatchD scan R	7	100,00%	100,00%
LatchD scan S	3	100,00%	100,00%
LatchD scan RS	3	100,00%	100,00%
NCL1	30	100,00%	100,00%
NCL2	43	100,00%	100,00%
NCL3	39	100,00%	100,00%
SR latch	2	100,00%	100,00%
Two Port Latch	5	100,00%	100,00%
Two Port Latch R	5	100,00%	100,00%
Two Port Latch S	3	100,00%	100,00%
Two Port Latch RS	6	100,00%	100,00%
XOR FF	4	100,00%	100,00%
XOR FF R	8	100,00%	100,00%
XOR FF S	8	100,00%	100,00%
XOR FF RS	4	100,00%	100,00%
XOR Latch	3	100,00%	100,00%
XOR Latch R	4	100,00%	100,00%
XOR Latch S	4	100,00%	100,00%
XOR Latch RS	3	100,00%	100,00%

5.2 SPICE validation

After validating the cells with a logical model, it is interesting to validate different topologies. Using the previous application and using the SPICE simulator, the generator sequence was implemented and a template with the expected output. Each input and each expected output was implemented as a (Piece-wise Linear) PWL source and compared with a XOR gate. The expected output was previously calculated with the sequence and the CUT model. Two types of cell were used, a Type D Flip-Flop with Asynchronous Set and Reset and Type D latch with Asynchronous Set and Reset. The flip-flops had two topologies and the latch one.

5.2.1 Nor Type D Flip-Flop with Asynchronous Set and Reset

Figure 5.2: Flip Flop topology



Source: The author

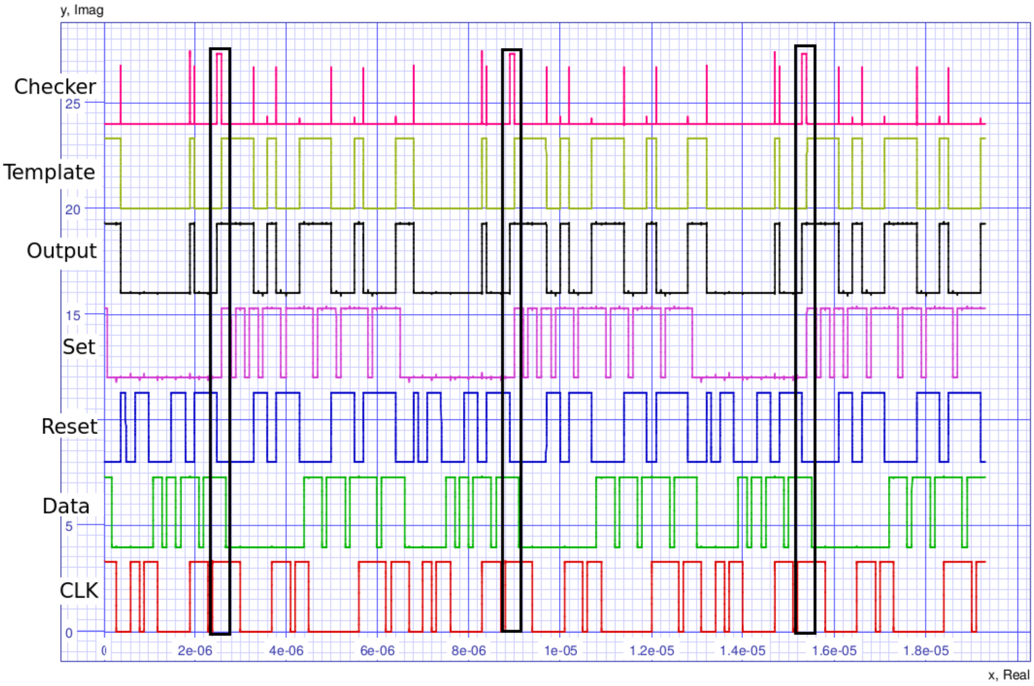
This first cell to be validated implements a flip-flop, whose topology is different from the usual master-slave latch topology. This one is made using NOR-gates as shown in Figure 5.2. Using 3 instance previously calculated from the application created on Section 5.1.2, a SPICE simulation was made in order to validate this topology. Table 5.6 shows the input setup for the three of the instances respecting the generator output ordering.

Table 5.6: Instances Input Setup

Instance	Input Setup	Negated Inputs
0	Reset, Data, Set,Clock	Reset, Data, Clock
1	Reset, Data, Set,Clock	Set
2	Reset, Data, Set,Clock	Set, Clock

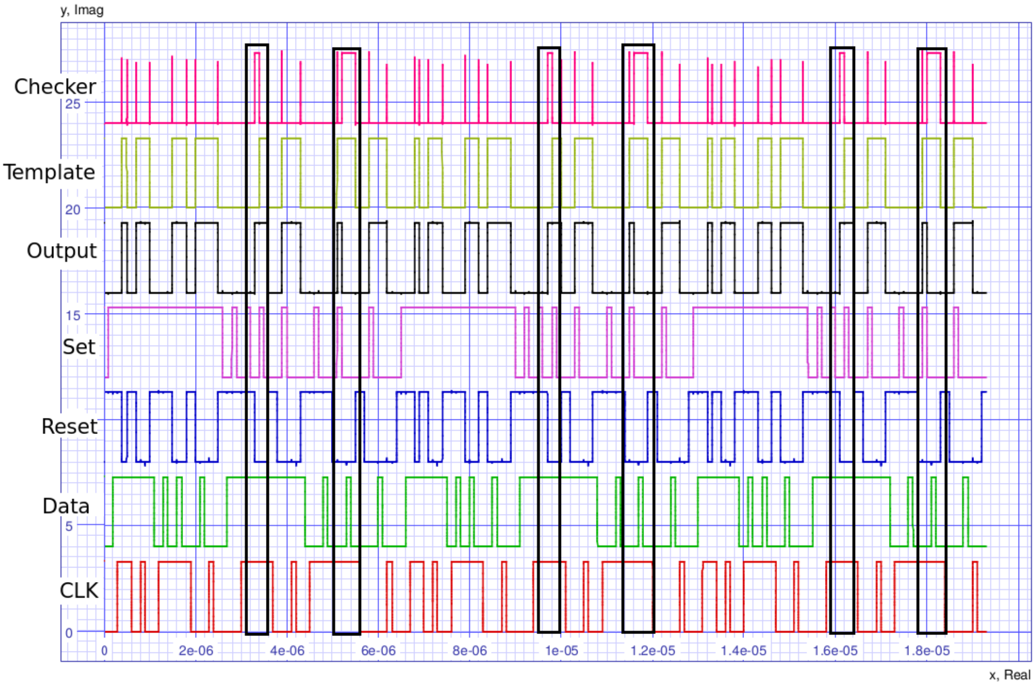
Using the setup with 3 instances of the flip-flop, it was discovered that because of the flip-flop topology, when the "Clock" input is "1", the "Reset" or the "Set" signal changes to the "0" logical value, the value on the "Data" input is transmitted to the output, indicated by the black squares. This is an erroneous behavior. Figure 5.3, Figure 5.4 and Figure 5.5 show the waveforms for the 3 instances. This is a proof why, testing the transitions is important.

Figure 5.3: SPICE simulation of the first instance flip-flop.



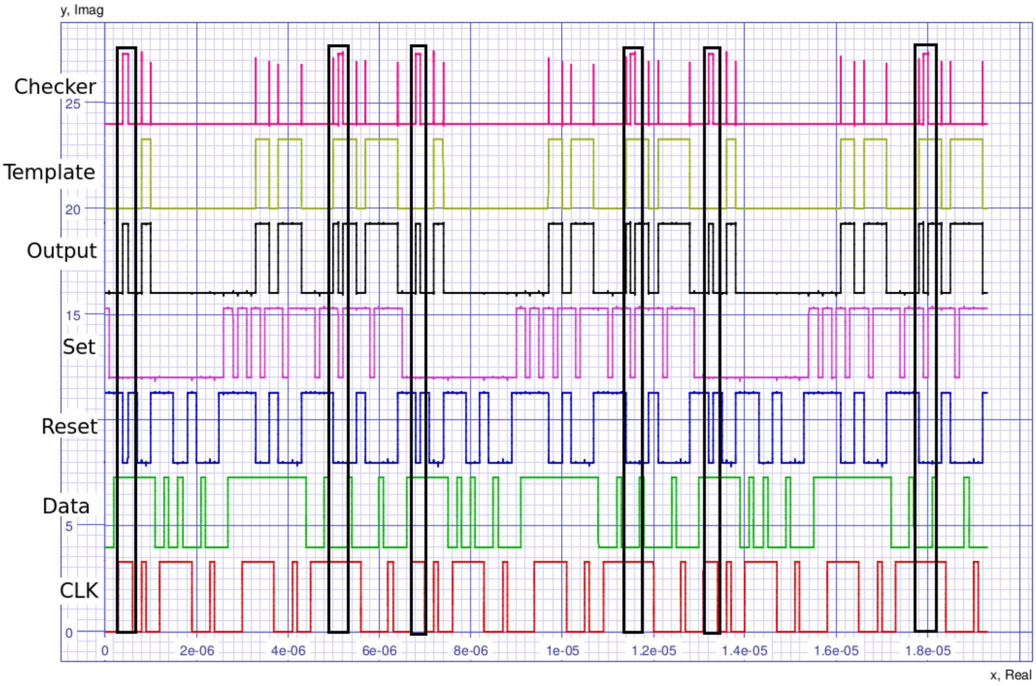
Source: The author

Figure 5.4: SPICE simulation of the second instance flip-flop



Source: The author

Figure 5.5: SPICE simulation of the third instance flip-flop

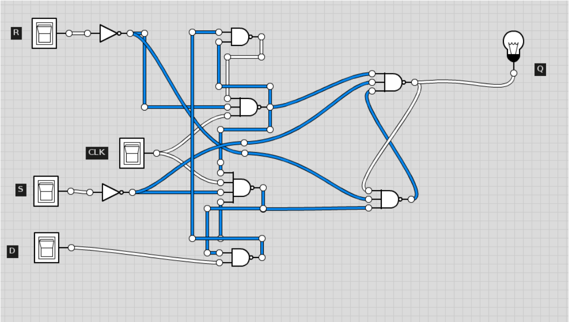


Source: The author

5.2.2 NAND Type D Flip-Flop with Asynchronous Set and Reset

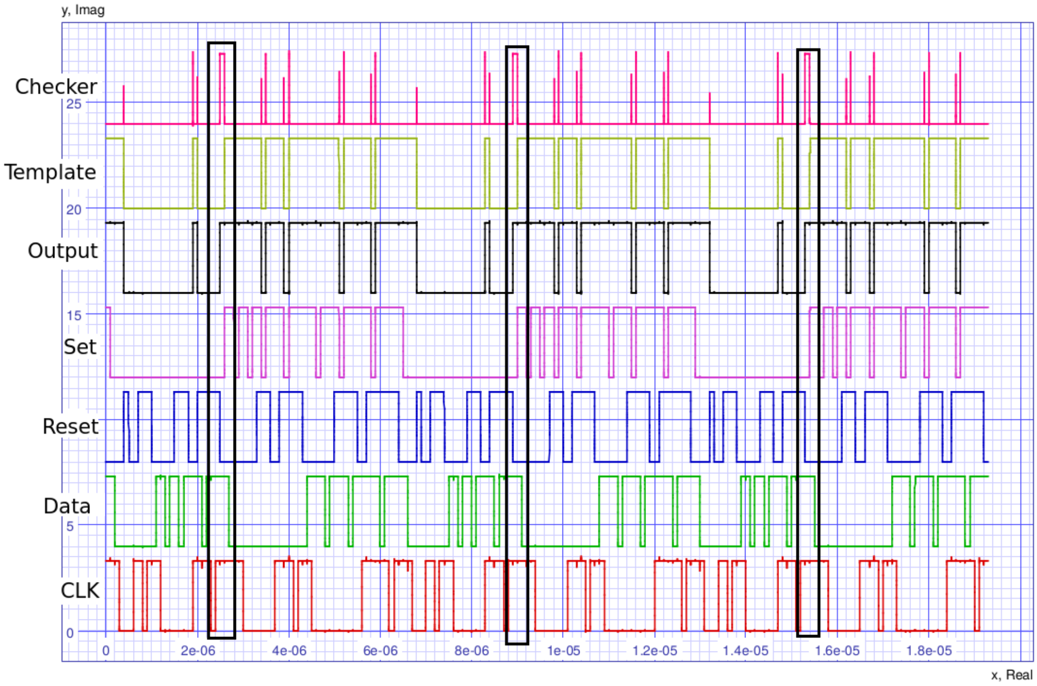
Like the previous cell, this one also implements a Flip Flop with asynchronous set and reset but with NANDs, as shown in Figure 5.6. Another difference is that when both set and reset are on at the same time set takes precedence before reset. The same instances as the previous example were used since they have the same theoretical model. Figure 5.7, Figure 5.8 and Figure 5.9 show the resulting simulation. As the previous topology present the same error as indicated by the black squares.

Figure 5.6: Flip Flop topology



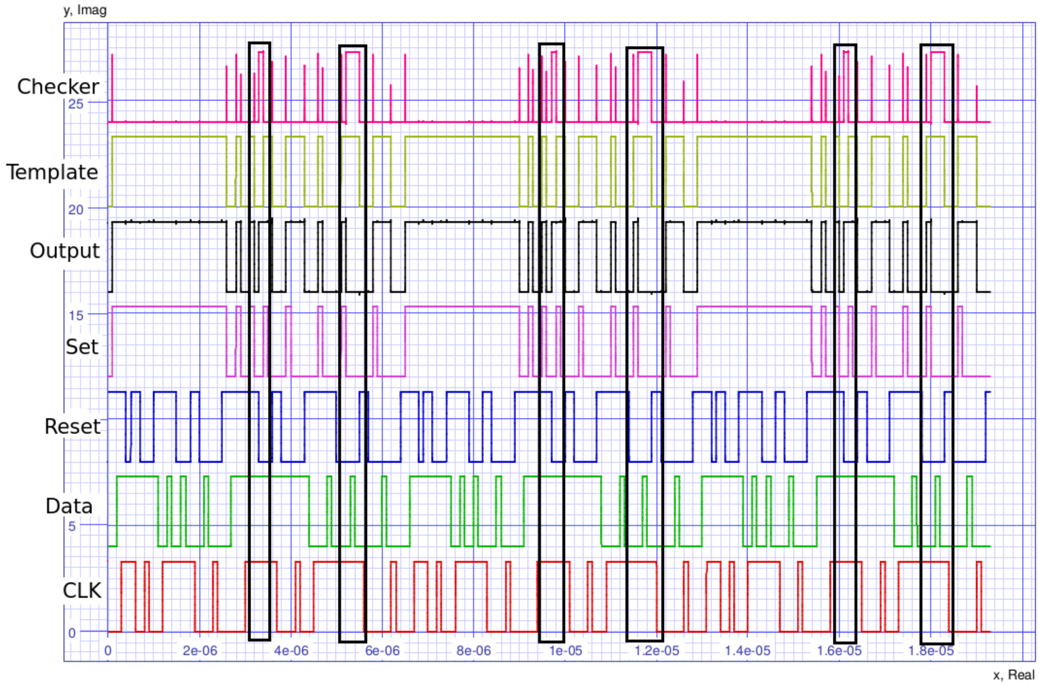
Source: The author

Figure 5.7: SPICE simulation of the first instance flip-flop.



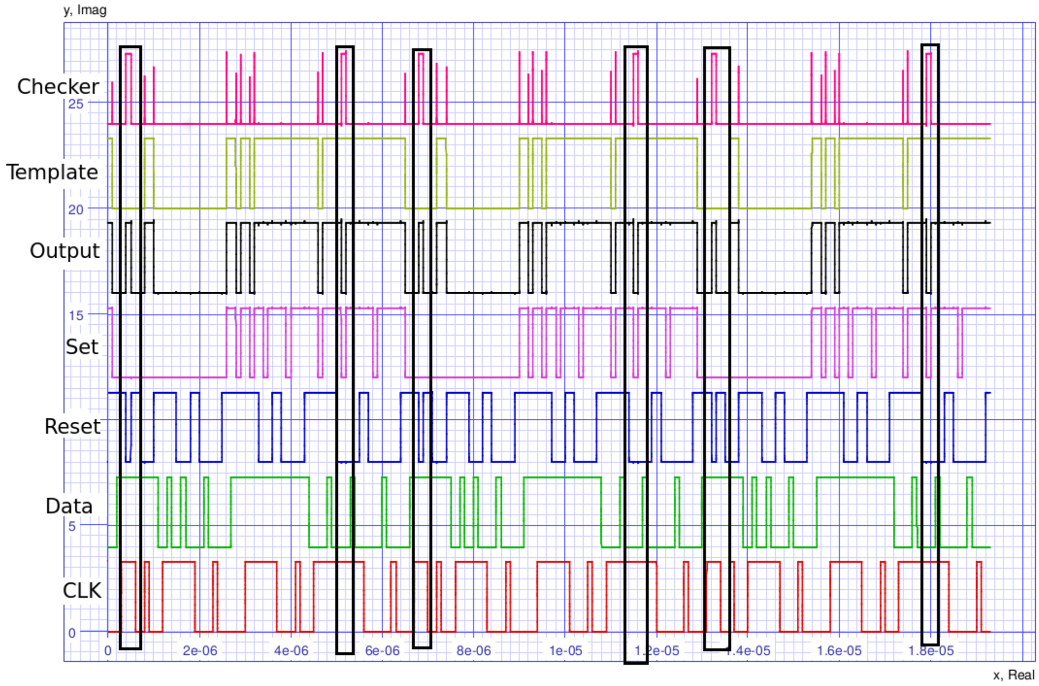
Source: The author

Figure 5.8: SPICE simulation of the second instance flip-flop



Source: The author

Figure 5.9: SPICE simulation of the third instance flip-flop



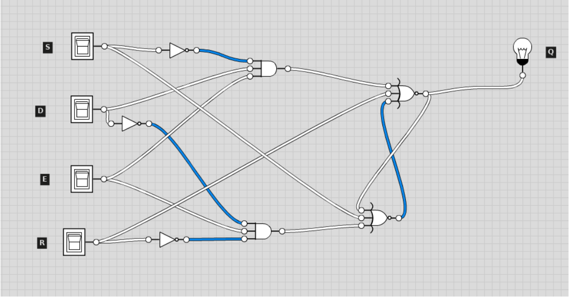
Source: The author

5.2.3 Single Gate D Type Latch with Asynchronous Set and Reset

Table 5.7: Instances Input Setup

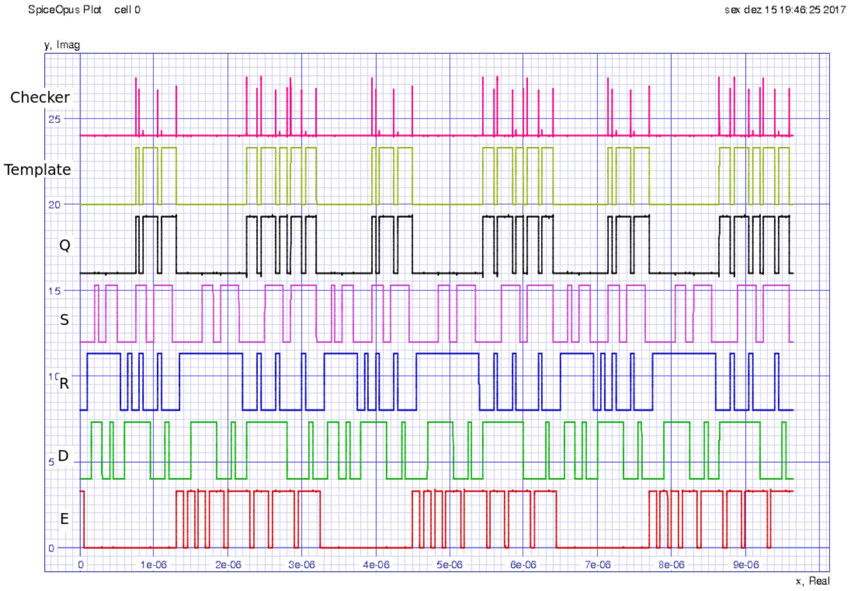
Instance	Input Setup	Negated Inputs
0 & Reset	Enable, Set, Data, Reset	Enable
1 & Reset	Enable, Set, Data, Reset	Set
2 & Reset	Enable, Set, Data, Reset	Data

Figure 5.10: First Latch Topology



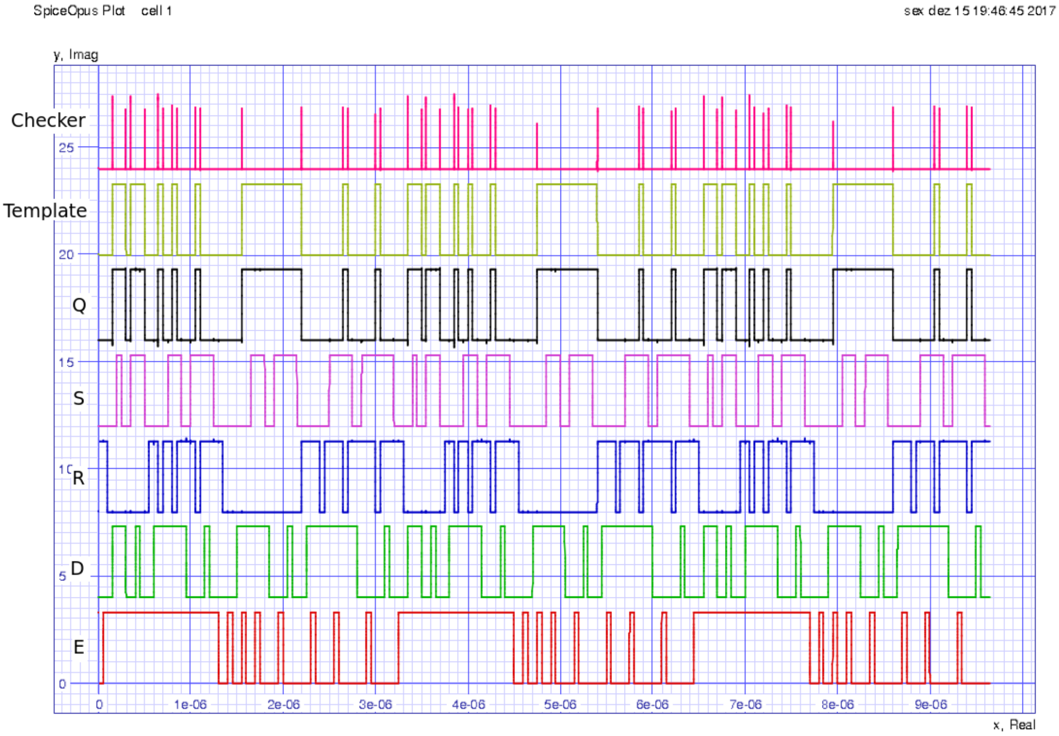
In the previous experiment, it was demonstrated the capacity of detecting errors. However, it is important to see if a correct topology will pass the validation. The topology tested came from the original topology shown in Figure 5.10. In order to shown the original functioning, it was also validated using this application. Table 5.7 shows the input ordering and which one was negated. Figure 5.11, Figure 5.12 and Figure 5.13 show the resulting waveforms after simulating.

Figure 5.11: SPICE simulation of the first instance of the first topology.



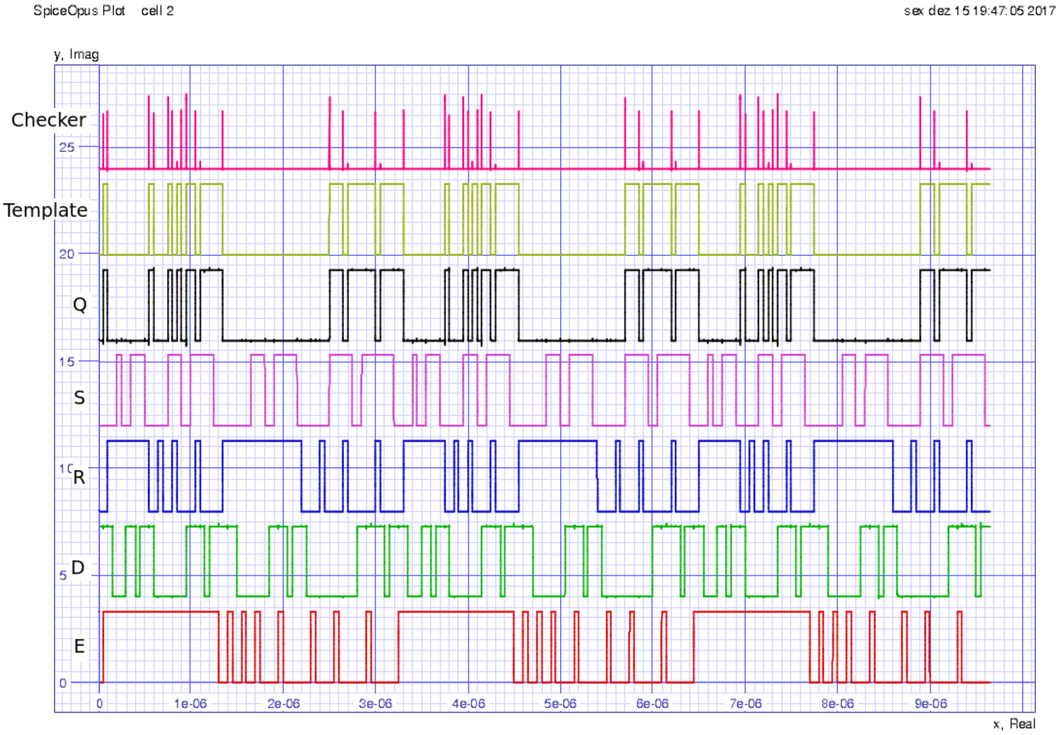
Source: The author

Figure 5.12: SPICE simulation of the second instance of the first topology



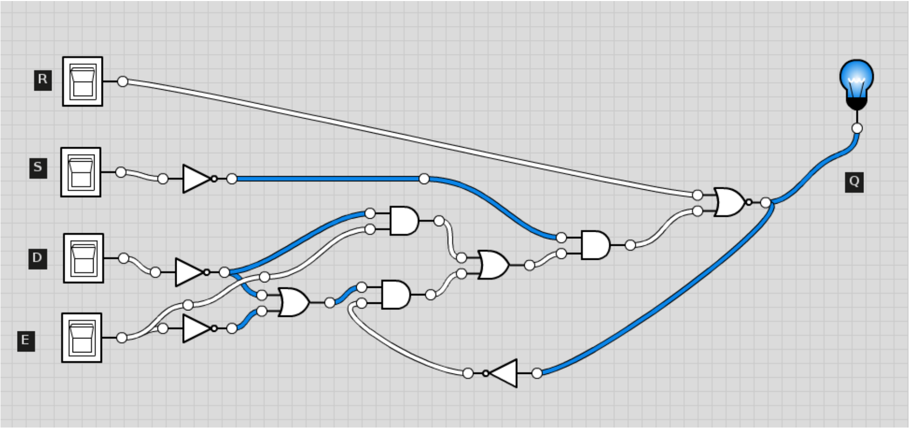
Source: The author

Figure 5.13: SPICE simulation of the third instance of the first topology



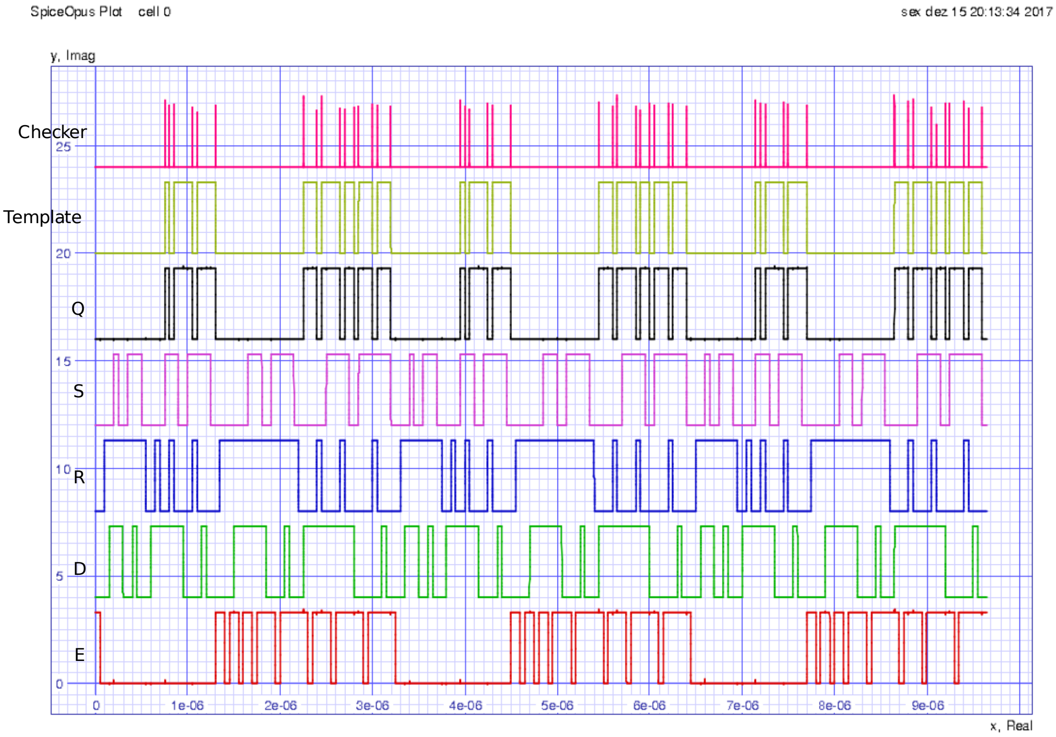
Source: The author

Figure 5.14: Second Latch Topology



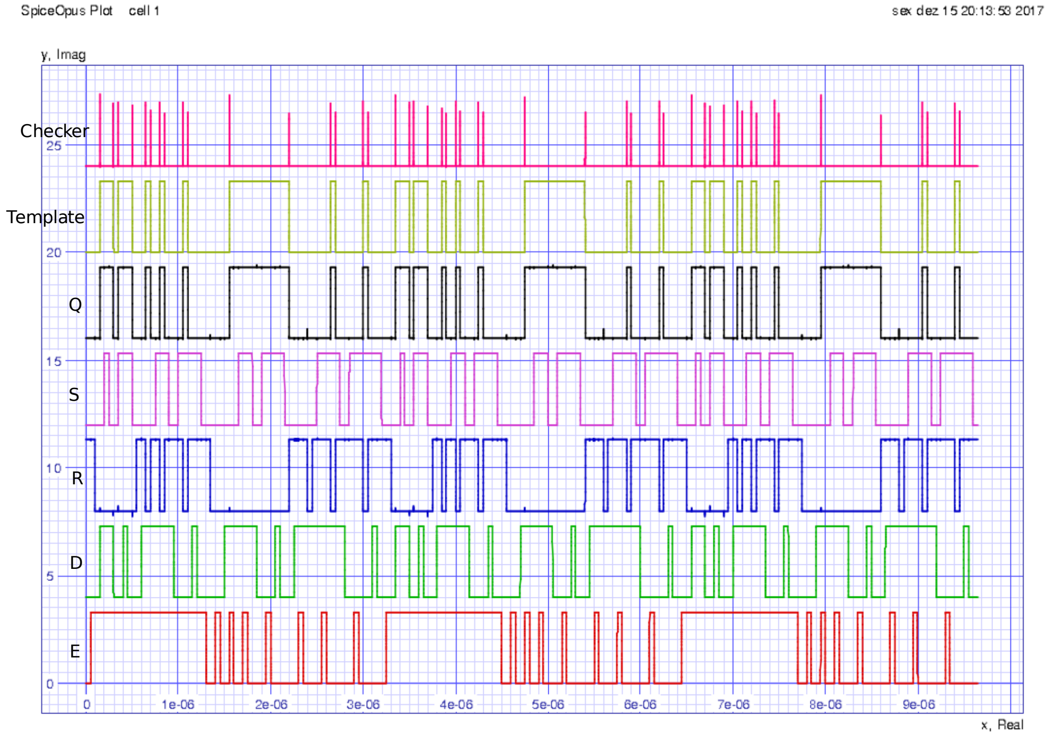
Seeing that the first topology is functioning correctly, the second topology must be functioning properly. Using DeMorgan and other Boolean properties the single gate topology shown in Figure 5.14 can be created from the topology in Figure 5.10. Again, it was simulated with the same instances found in Table 5.7. Figure 5.15, Figure 5.16 and Figure 5.17 show the resulting waveforms after simulating. As expected, the second topology is functioning with the expected behavior, therefore the transformation from the first topology was made without errors.

Figure 5.15: SPICE simulation of the first instance of the second topology.



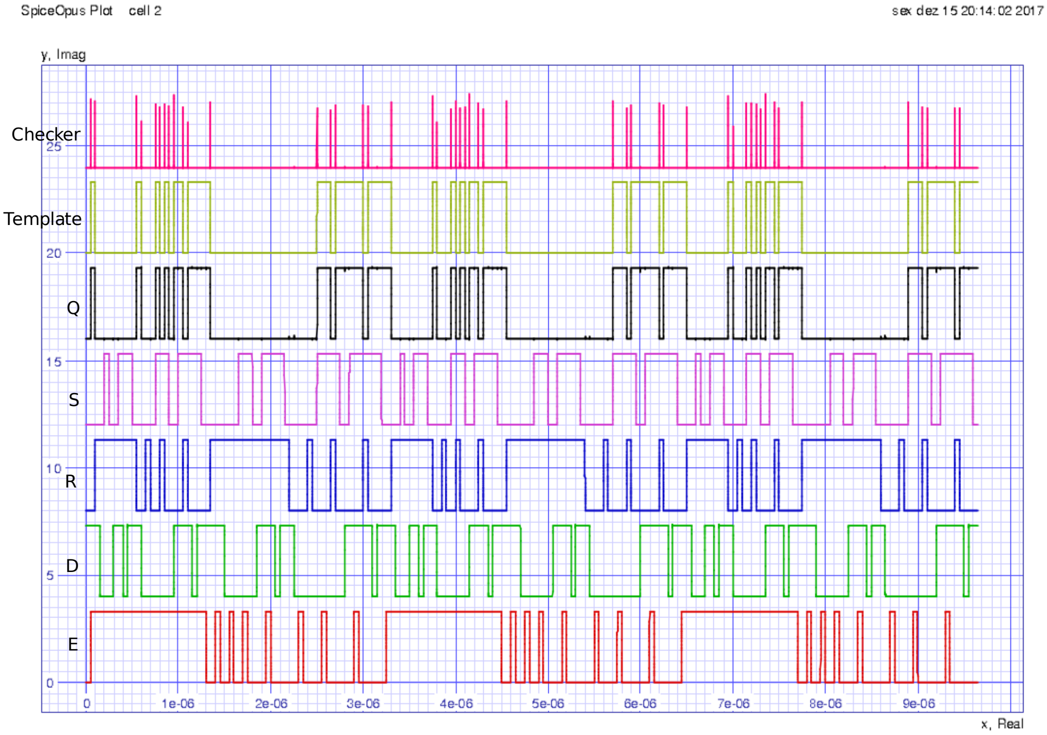
Source: The author

Figure 5.16: SPICE simulation of the second instance of the second topology



Source: The author

Figure 5.17: SPICE simulation of the third instance of the second topology



Source: The author

6 CONCLUSÕES

In this work, a test pattern generator was proposed. The proposed generator attends all specified requirements. It was proven to be generic, cyclic and transitionate only one bit per step. Since it covers all possible input combination and all transitions, it produces a sequence with a length of $N * 2^N$, for a cell with N inputs.

In the context of sequential cells validation, the generator proved to be effective, having a high coverage. This was achieved using some modifications in order to keep the generator generic. Only one cell had total coverage when using a single instance, the other required a few instances. When validating in hardware, some instances proved to not behave in a cycle and may raise a false positive error, therefore it was proposed to disregarding the coverage of the first cycle. This method only slightly decrease the coverage of some cells and others had a slight increase of instances necessary to keep total coverage. In the context of validating in SPICE, the generator was proved to be easily adaptable to this environment.

This generator model was published in two previous articles on for Simpósio Sul de Microeletrônica (SIM) and the other for the journal Revista Jr de Iniciação Científica em Ciências Exatas e Engenharia (ICCEEg) annexed to this work. In the ICCEEg article only the generators model was discussed.

REFERENCES

AGATSTEIN, W.; MCFAUL, K.; THEMINS, P. Validating an asic standard cell library. In: **Third Annual IEEE Proceedings on ASIC Seminar and Exhibit**. [S.l.: s.n.], 1990. p. P12/6.1–P12/6.5.

ALIOTO, M.; CONSOLI, E.; PALUMBO, G. Analysis and comparison in the energy-delay- area domain of nanometer cmos flip-flops: Part i—methodology and design. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, June 2011.

AVELAR, H. H.; BUTZEN, P. F.; RIBAS, R. P. Automatic circuit generation for sequential logic debug. In: **2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)**. [S.l.: s.n.], 2015. p. 141–144.

BUSHNELL, M.; AGRAWAL, V. **Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits**. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 1475781423, 9781475781427.

DORAN, R.; SCIENCE, C. for D. M. . T. C. **The Gray Code**. Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, 2007. (CDMTCS research report series). Available from Internet: <https://books.google.com.br/books?id=ff25jwEACAAJ>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728.

MAKAR, S. R.; MCCLUSKEY, E. J. Checking experiments to test latches. p. 196–201, Apr 1995. ISSN 1093-0167.

RABAEY, J. M. **Digital Integrated Circuits: A Design Perspective**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN 0-13-178609-1.

RIBAS, R. et al. Ring oscillators for functional and delay test of latches and flip-flops. 08 2011.

RIBAS, R. P. et al. Circuit design for testing standard cell libraries. **WCAS 2011, Workshop on Circuits and System Design**, v. 1, 2011.

RIBAS, R. P. et al. Self-checking test circuits for latches and flip-flops. In: **2011 IEEE 17th International On-Line Testing Symposium**. [S.l.: s.n.], 2011. p. 210–213. ISSN 1942-9398.

SPARS, J.; FURBER, S. **Principles of Asynchronous Circuit Design: A Systems Perspective**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441949364, 9781441949363.

TRAN, L. et al. **Null convention logic (NCL) based asynchronous design — fundamentals and recent advances**. [S.l.: s.n.], 2017. 158-163 p.

WESTE, N.; HARRIS, D. **CMOS VLSI Design: A Circuits and Systems Perspective**. 4th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0321547748, 9780321547743.

Design Strategy for Testing Sequential Logic Gates Based on Signal Pattern Generator

Pablo Rafael Bodmann, Renato Perez Ribas

Abstract—**The validation of standard cell libraries used on digital integrated circuit design is a crucial task. However, the test of sequential logic gates is quite complex due to the inherent memory effect. In this work, it is proposed a universal signal pattern generator to be applied in a novel test circuit strategy. To model the problem our approach creates a pattern sequence over structures like graph and tree structures. As proof-of-concept, a Java application has been developed. Experimental results have shown that such a strategy has attained a high coverage of logic faults.**

Index Terms—**digital circuit, standard cell library, logic gate, sequential cell, test.**

I. Introduction

THE design of integrated circuits (IC) comprises many tasks. In order to reduce design costs and time-to-market, standard cells design methodology has been widely adopted. This methodology is based on the reuse of blocks and small circuits (named as cells) that implement logic functions. These pre-designed cells are available in a library and must be pre-evaluated and pre-validated before using in ASIC design.

Since a library usually comprises a large number of logic gates and all of them must be validated, efficient test setups are essential to reduce design costs. For combinational gates, we consider that the solution proposed in [1] is quite simple and effective. However, the test of sequential logic gates is more complex than the combinational ones due to the inherent memory behavior, *i.e.*, the current output signals depend not only on the current input variables but also on the previous sequence of these ones. Another difficulty is the presence of asynchronous signals that have priority in the sequential behavior.

Several attempts have been proposed related to the testing of sequential logic gates. In [2], the authors propose the use of Boolean equation describing the gate and create a corresponding state table in order to calculate the minimum input sequence to cover all possible defects usually observed in the registers. Despite of having high coverage, this solution is not general, *i.e.*, the sequence depends on the specific circuit behavior targeted. This is a huge problem when testing a large set of gates with different behaviors because for each group of gates a single generator must be created so increasing the complexity of the test. Another approach evaluates latches and flip-flops using shift-register and counter circuitries, respectively [3]. However, it is not shown the way to create a new shift-register for testing other sequential logic gates different from the ones treated in that work. Therefore, this solution is not general yet. An approach based on finite state machine, for describing the sequential gate behavior, is found in [4]. However, as this approach also represents particular solutions for specific sequential gates, a large circuit area overhead is expected.

In this work, we proposed a universal logic vector generator for testing sequential circuits. The main idea is that the generator provides one signal transition per cycle, covering all possible steady states and signal transitions. Repeated output signal transitions do not occur.

This article is organized as follows. Section II discusses the possible static states, expected and unexpected transitions of sequential logic gates. Section III analyzes some previous approaches related to the testing of sequential cells. Section IV presents the proposed approach. Section V shows the circuit generator implementation. Section VI shows and discusses some experimental results. The conclusions are outlined in Section VII.

II. Preliminaries

This section presents the logic behavior observed in sequential logic gates such as the steady states and dynamic states (expected and unexpected transitions). Three basic gates are taken into account to illustrate these situations: C-element (Mller cell), D-type latch and D-type flip-flop, both with asynchronous reset signal. In the C-element circuit, when both inputs are equal the same logic value is presented at the output, and when the inputs are different the gate output keeps its previous state [5]. In the case of D-type latch with asynchronous reset, it is a level sensitive logic gate, *i.e.*, when the enable signal is high the value at the data input is transmitted to the output, and when the enable input is low the previous output is maintained. The D-type flip-flop, on the other hand, has a similar behavior to the latch but it is border sensitive circuit, *i.e.*, the value at the data input is only transmitted to the output when the clock input rises.

A. Steady States

In order to have the maximum test coverage, the analysis of the steady states of the circuit is a crucial task. The steady states are the combination of inputs and output values. Table I shows the steady states of the C-element, Table II presents the steady states of the D-type latch with asynchronous reset signal, and Table III shows the steady states of the D-type flip-flop with asynchronous reset.

It is worth to note that some input combinations can have two possible outputs. The reason is the memory effect of the

Pablo Rafael Bodmann and Renato Perez Ribas are with the Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), 9500 Porto Alegre, Brazil (e-mail: {prbodmann, rpribas}@inf.ufrgs.br).

Research partially funded by CNPq Brazilian agency

TABLE I
C-element steady states

A	B	Previous Q	expected Q
0	0	X	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	X	1

TABLE II
D-type latch with asynchronous reset steady states

R	D	E	Q
0	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	1	0	1
0	1	1	1
1	X	X	0

TABLE III
D-type Flip-Flop with asynchronous reset steady states

R	D	CLK	Q
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	X	X	0

latch. In the flip-flop, it still is more significant. The reason for that is the fact that the latch is a level sensitive gate whereas the flip-flop is a border sensitive circuit. Therefore, the flip-flop presents more states to be covered. In order to cover these states, it is necessary to pass through a specific transition. The states with reset with the ”1” logic value were omitted because the output is stuck at the 0 logical value. The C-element has an interesting behavior compared with latch and flip-flop gates. Instead of having the memory behavior controlled by a single input, in this case, it is controlled by both of them. This behavior is desired in asynchronous circuit, where there is no global clock signal [5].

B. Dynamic States: Expected Transitions

Another important aspect of test coverage is the expected transitions, *i.e.*, when an input changes there is a transition at the output. Table IV shows the expected transitions of the C-element gate, Table V presents the expected transitions of the D-type latch, and Table VI shows the expected transitions of the D-type flip-flop.

Again, the difference among these circuits can be observed. The transition in the data input is only propagated to the output when the enable input is high or when the enable input rises the data input and this one differs from the current state of the latch. This propagation only occurs in the flip-flop when the clock signal rises and the data input differs from the current state of the flip-flop. The propagation of transitions in the C-element occurs when one input transitions to a value equal to the other, and this new input value differs from the current C-element state.

TABLE IV
C-element expected transitions

A	B	Previous Q	expected Q
1	↑	0	↑
↑	1	0	↑
↓	0	1	↓
0	↓	1	↓

TABLE V
D-type latch with asynchronous reset expected transitions

R	D	E	Previous Q	Q
0	0	↑	1	↓
0	↑	1	0	↑
0	↓	1	1	↓
↑	1	1	1	↓
↑	1	0	1	↓
↓	1	1	0	↑

TABLE VI
D-type flip-flop with asynchronous reset expected transitions

R	D	CLK	Previous Q	Q
0	1	↑	0	↑
0	0	↑	1	↓
↑	X	X	1	↓

C. Dynamic States: Non-Expected Transitions

Besides testing expected transitions, it is important to test the non-expected transitions. These transitions occur when the input is transitioned but the output must stay stable. Table VII shows the non-expected transitions for C-element, Table VIII presents the non-expected transitions for latch, and Table IX shows the ones for the flip-flop.

The states show the memory effect of these circuits, when a transition does not propagate to the output. These transitions must be covered in order to completely test whether the gate is holding the state and not transitioning.

III. Related Works

As mentioned in the Introduction section, there are proposed works which show some approaches to test sequential cells, especially D-type latches with asynchronous set and reset and D-type flip-flop with asynchronous set and reset.

At [2], it is proposed a set of necessary conditions in order to fully test latches. Using a logical equation to describe the possible states of the gate, the paper delimiters some essential

TABLE VII
C-element non-expected transitions

A	B	Previous Q	expected Q
↑,↓	0	0	0
↑,↓	1	1	1
0	↑,↓	0	0
1	↑,↓	1	1

TABLE VIII
D-TYPE LATCH WITH ASYNCHRONOUS RESET NON-EXPECTED TRANSITIONS

R	D	E	Previous Q	expected Q
0	↑,↓	0	X	X
1	↑,↓	X	X	X
1	X	↑,↓	X	X
↑,↓	X	X	0	0
0	0	↑,↓	0	0
0	1	↑,↓	1	1

TABLE IX
D-TYPE FLIP-FLOP WITH ASYNCHRONOUS RESET NON-EXPECTED TRANSITIONS

R	D	CLK	Previous Q	expected Q
0	↑,↓	X	X	X
0	X	↓	X	X
↑,↓	X	X	0	0

sequences that must be part of the checking experiment in order to cover all possible steady states. Despite detecting all faults of a logic gate, this solution misses some possible transitions, as well as it is not generic and it is not cyclic, requiring a reset signal.

At [3], the testing of D-type latch with asynchronous set and reset is done by instantiating a 12-bit shift-register with the same gate under test, as shown in Fig. 1. However, some latches have the set and reset signals behavior determined by the current overall state of the register, and the enable polarity signal is inverted at every couple of latches. The steady state coverage is almost 100% but the state where both set and reset are on is not covered. Moreover, some possible and unexpected output transitions are not covered by this approach.

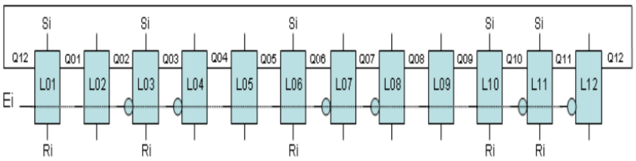


Fig. 1. Shift register setup described at [3]

For testing the D-type flip-flop with asynchronous set and reset, a modified 5-bit counter is created where each bit is the same gate under test, as shown in Fig. 2. Similar to the test of the latch, the set and reset signals of several bits are dependent of different states of the counter and are calculated by a handshake circuit. The test covers all steady states excluding the ones with both set and reset signals activated. The transition coverage is 50% of the unexpected transitions. Both solutions are specific either latch or to flip-flop, and it is not shown the way to create a shift-register to test other kind of sequential gates.

Another approach is defined at [4], where finite-state machine (FSM) is created using the circuit behavior description. The FSM passes through all steady states and signal transitions in order to create an input pattern sequence that has 100% of coverage. Despite being similar to this work, the mentioned generator is dependent of the gate behavior and the proposed one is dependent only to the number of signals. Another problem is that the length of the sequence varies with the initial state. This solution, despite of having 100% of coverage, is not general aw well as it is not cyclic requiring a reset signal.

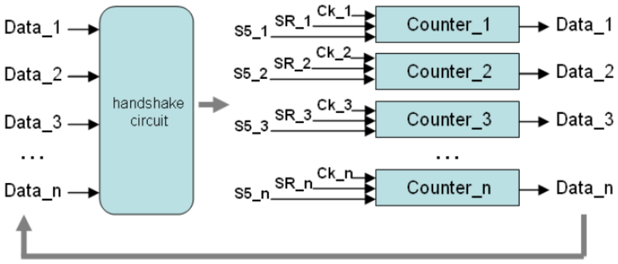


Fig. 2. Modified counter described at [3]

IV. Proposed Self-Check Setup

In this paper, it is proposed a universal generator for testing sequential logic gates, *i.e.*, a generator independent from the circuit behavior. The generator provides signals in a cyclic sequence, *i.e.*, a sequence that starts and terminates at the same input vector. This is desired because the generator can be left running to stops only when an error occurs. It covers also all input states and signal transitions. The signal transitions occur by changing one bit per step. Such a characteristic is important because it avoids timing race conditions that can cause metastability and raise false-positive errors. Another reason is that it is easier to debug when an error occurs.

This work is very similar to [2] and [4] due to the use of FSM strategy. The difference is that we disregard the gates behavior and only focus the number of inputs that the circuits have. Another difference between these two previous works and the one proposed is that this generator is cyclic and can run multiple times without external signals if necessary.

The proposed generator is part of a larger test bench. Fig. 3 shows the proposed generator and the test bench. Using the concept of self-checking, *i.e.*, the test bench does not need external clock signal to run. Instead, it creates its own temporizing signal. When using this principle, the generator must also provide a template that is compared with the gate output. Since the template signal is usually faster than the circuit output, the checker provides a 0 logic value. If the output is correct, then the checker provides 1 logic value, creating a rising border at the internal clock signal. Thus, making the generator provides the next states. If an error occurs, this rising edge does not occur, locking the generator at the current state.

V. Generator Model

A. Modeling

Since the proposed generator is universal, its behavior must be independent from the gate under test. Therefore, we ignore the gate memory effect and treat it as a black box. One

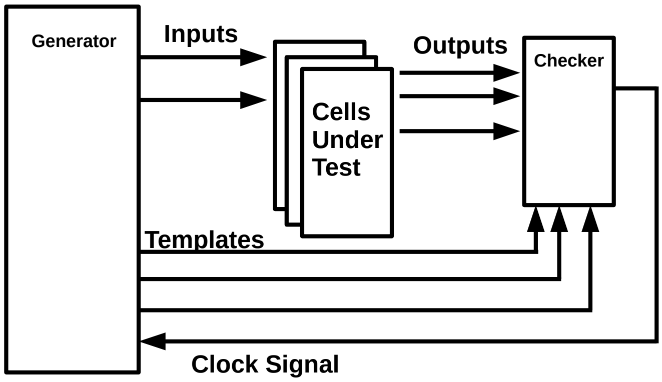


Fig. 3. The self-check setup

candidate would be the Gray code but it does not cover all possible states, as shown in Table X. Thus, a better model is necessary in order to solve such a deficiency.

TABLE X
3-BIT GRAY CODE AND ITS MISSING TRANSITIONS.

Gray Code	Missing Transitions
000	000 → 010 and 000 → 100
001	001 → 101 and 001 → 000
011	011 → 001 and 011 → 111
010	010 → 011 and 010 → 000
110	110 → 010 and 110 → 100
111	111 → 110 and 111 → 011
101	101 → 111 and 111 → 001
100	100 → 101 and 100 → 110

Using the content in Table X, a graph can be built. The nodes represent the possible states and the edges of the transitions. Fig. 4 shows the resulting graph of a 3-input gate. This kind of graph is called an n-cube graph or a hypercube. Since it is interesting to cover both rising transition, when a bit goes from logic value 0 to value 1, and falling transition, when a bit goes from logic value 1 to value 0, the graph must be bidirectional.

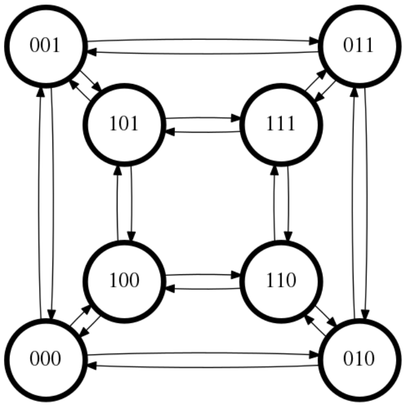


Fig. 4. 3-Dimensional Hypercube representing a 3 bit Gray Code

Once having built the graph, it is only necessary to find out an Euler cycle, *i.e.*, a cycle that passes through all edges exactly once, beginning and ending at the same node. Due to the form of the graph, it has many different Euler cycles and to find one, it is proposed a simple solution. The graph is transformed in a tree. Each node can be interpreted as a number and each node must have their sons with larger values and their father with smaller value. The root will have the lowest value possible, the state with every input at the 0 logic value. In order to preserve all possible transitions, some nodes must be repeated. Fig. 5 shows the resulting tree for the graph in Fig. 4. Once having the tree, it is only necessary to make a depth first search (DFS) in order to generate an Euler cycle.

B. Implementation

During the implementation, some shortcuts can be used in order to speed up the process. The graph building phase can be skipped and the tree can be represented with a table, such as shown in Table XI. The columns are the nodes and each row has the possible next node. The creation of this table can be done by flipping the bits with 0 in order to save only the numbers larger than the current. Fig. 6 shows the pseudocode for the creation of data shown in Table XI. The DFS can be performed by saving the current column, jumping to the first son and marking it as visited. If jumping to a column with no sons or with all sons visited, the algorithm jumps back to the previous node from which it came. The algorithm stops when all sons from the 0 state are visited. Fig. 7 shows the pseudo algorithm for the creation of the sequence.

C. Complexity

With a gate with N inputs, the possible states are 2^N different states. In order to calculate the larger values, each bit up to the *Nth* bit must be tested and inverted if necessary. Thus, the complexity for creation of data in Table is $N * 2^N$. The creation of the sequence is also $N * 2^N$. That is, in order to represent all possible transitions, it is necessary to storage and pass through $N * 2^N + 1$ states. Since the last state is the 0 state, it can be ignored and, in order to maintain this last transition, it is only necessary to return to the beginning of the list.

VI. RESULTS

In order to validate the proposed universal signal generator, a Java application has been created. In this application, the behavior of the circuit under test was described and the corresponding steady states and signal transitions calculated automatically. Afterwards, the behavioral model of the logic gate was submitted to the sequence stimuli transitions and the test coverage was evaluated. The first steady state of the generator output is set at the logic value 0. The gates

TABLE XI
TABLE CONTAINING EACH NODE ADJACENCY LIST REPRESENTING THE TREE IN FIG. 5

Nodes	000	001	010	011	100	101	110	111
Next Nodes	001	011	011	111	101	111	111	
	010	101	110		110			
	100							

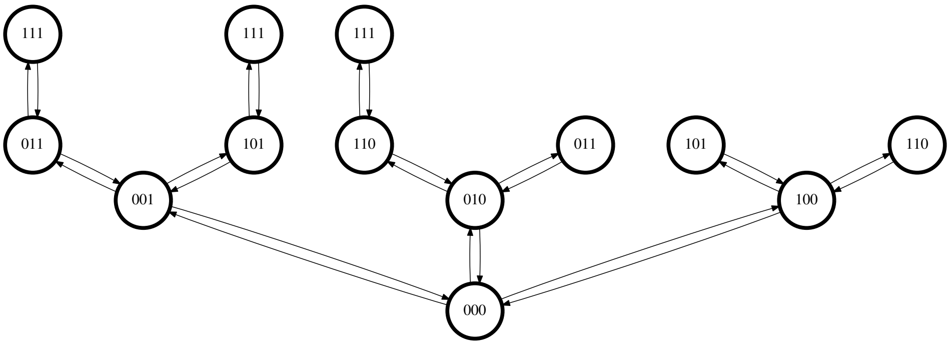


Fig. 5. The resulting tree from the graph in Fig 4

```
Input: The number of inputs of the CUT
procedure $Create\_Table(N)$
    $adjacency\_table = [[] * N]$
    for $i = 0; i < 2^N; i++$ do
        $mask = 1$
        for $j = 0; j < N; j++$ do
            if $temp \wedge mask == 0$ then
                $temp = i \oplus mask$
                $adjacency\_table[i].append(temp)$
            end if
            $mask \ll 1$
        end for
    end for
    return $adjacency\_table$
end procedure
```

Fig. 6. Pseudocode for creating Table of adjacency lists

```
Global pattern_list
Global adjacency_table
procedure $Create\_pattern(node)$
    pattern_list.append(node)
    for each element in adjacency_table[node] do
        $Create\_pattern$(element)
        pattern_list.append(node)
    end for
    mark_visited(node)
end procedure
```

Fig. 7. Pseudocode for creating the pattern sequence

used as case studies for such a validation was the D-type latch, D-type latch with asynchronous set, D-type latch with asynchronous reset, D-type latch with asynchronous set and reset, D-type flip-flop, D-type flip-flop with asynchronous set, D-type flip-flop with asynchronous reset, D-type flip-flop with asynchronous set and reset, and a Mller cell (or C-element). The generator outputs ordering of signal connections used was: output 0 is enable (E) or clock (Ck) inputs, output 1 is data (D) input, output 2 is reset (R) or set (S) input, and output 3 is set when reset signal is also available. Since there are vectors where set and reset are turned on, it was supposed that reset has priority over set, it means, when both set and reset asynchronous signals are activates the gate output goes down (value 0). Table XII shows the test coverage when considering a single instance of each sequential gate connected to the proposed generator.

TABLE XII
COVERAGE USING A SINGLE CELL

Cell	Steady State Coverage	Dynamic States Coverage
D-Latch	100%	75%
SD-Latch	100%	83.34%
RD-Latch	90%	80%
RSD-Latch	100%	90.27%
D-FF	87.5%	56.25%
SD-FF	91.67%	69.44%
RD-FF	83.34%	66.66%
RSD-Ff	100%	81.25%
C-element	100%	66.66%

Using a single instantiation of the circuit under test, 100% of coverage was not attained in some cases, neither in steady state coverage nor in dynamic coverage (transitions). It is resulting from the memory effect of sequential circuit. Some input to output signal transitions and states are hidden from the stimulus sequence of the generator. Fig. VI shows the modelling of a D-type flip-flop with asynchronous reset. The borderless green circle represents the possible input vectors that the corresponding output is expected to be 1, and the red border circle represents the input vector with the output in 0. When compared to the modelling in Fig. 4, it can be seen that some input vectors are repeated and there are more transitions, being some of them one-direction only.

Since it is desired to maintain the universal characteristics of the generator, the direct use of the circuit under test behavior cannot be used to create a specific sequence with higher coverage. Instead, the circuit behavior, the vector sequence and the multiple instantiations of the same gate can be previously calculated. This multiple instances can have none, one or more negated inputs, that means, the connection of the signal from the generator to the circuit is negated before arriving in the circuit input. Moreover, such a connection can be permuted, *i.e.*, not following the previous connection ordering. Table XIII shows the dynamic coverage using such a strategy. The number

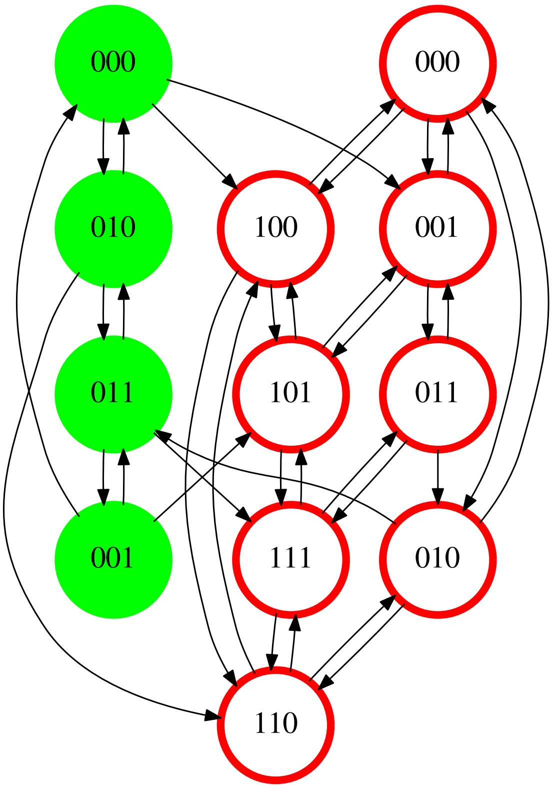


Fig. 8. D-type flip-flop with asynchronous reset model

of instances of the gate under test is the minimum necessary to achieve the maximum test coverage. By making so, all steady state coverage attained (100%), therefore it was omitted. As can be seen for almost all gates under test, the dynamic test coverage is 100%. It is possible because when permuting or negating the stimulus signals, the circuit is actually starting at a different state and passing through a different sequences without modifying the generator. In the case of the flip-flops without asynchronous signals (D-FF) and with reset signal (RD-FF), where the complete test coverage is not achieved, only a single dynamic state has not been observed, when clock signal is rising, data signal is at high value, reset is at low value, and the current output state is high and does not change.

In order to evaluate the scaling factor of the proposed approach in terms of circuit area, it is considered two possible designs for the generator: the first one by synthesizing the circuit from a Verilog description of the universal generator; the second one by applying a ROM block with the planned signal sequence. In the synthesis solution, the logic gates used were been restricted to only 2-input NOR (NOR2), inverter and flip-flop in order to estimate the resulting circuit size in equivalent gate metric, defined as a NOR2-based circuit area. Fig. 9 shows the number of logic gates used in the resulting map per number of outputs of the generator. As can be seen, the number of NOR2 and inverters grows exponentially with the number of bits at the output signal vector. The number of flip-flops grows linearly because they are only required in the counter circuit and in the output register.

TABLE XIII
COVERAGE USING MULTIPLE INSTANCES

Cell	# of cells used	Dynamic State Coverage
D-Latch	2	100%
SD-Latch	3	100%
RD-Latch	4	100%
RSD-Latch	2	100%
D-FF	3	93.75%
SD-FF	4	100%
RD-FF	5	97.23%
RSD-Ff	3	100%
C-element	2	100%

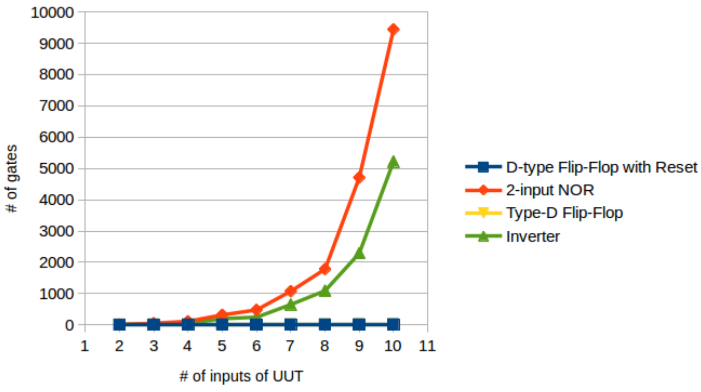


Fig. 9. Graph showing the numbers of gates per generator

The other possible physical implementation is the use of a read-only-memory (ROM) block. Since each signal state (output of generator) uses one line of the ROM, the resulting memory must be capable of mapping at least $N{*}2^N$ addresses. In order to profit of the full capacity of the memory, the sequence can be parted in N banks of memory with capacity for 2^N addresses. As in the previous design solution, through standard cells synthesis, this one grows exponentially as well.

VII. Conclusion

In this paper was proposed a general signal generator for testing standard cell libraries, in particular sequential logic gates (latches and flip-flops). Due to the inherent memory effect of these gates, even providing all possible single signal transition as stimuli, it was proven to be not sufficient in some cases to attain 100% of test coverage. On the other hand, the same generator can be applied to test several logic gates in parallel, reducing the area overhead. The generator was implemented in Java language, as proof-of-concept. The physical implementation of corresponding circuit is on progress.

References

[1] R. Ribas, S. Bavaresco, N. Schuch, V. Callegaro, M. Lubaszewski, and A. Reis, “Contributions to the evaluation of ensembles of combinational

logic gates," *Microelectronics Journal*, vol. 42, no. 2, pp. 371 – 381, 2011.

[2] S. R. Makar and E. J. McCluskey, "Checking experiments to test latches," pp. 196–201, Apr 1995.

[3] R. P. Ribas, Y. Sun, A. I. Reis, and A. Ivanov, "Self-checking test circuits for latches and flip-flops," in *2011 IEEE 17th International On-Line Testing Symposium*, July 2011, pp. 210–213.

[4] H. H. Avelar, P. F. Butzen, and R. P. Ribas, "Automatic circuit generation for sequential logic debug," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, Dec 2015, pp. 141–144.

[5] J. Spars and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*, 1st ed. Springer Publishing Company, Incorporated, 2010.

Test Pattern Generator for Latches and Flip-Flops

Pablo Rafael Bodmann[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

prbodmann@gmail.com

Abstract. *This article describes a proposition for a term paper. In this paper it is described a test pattern generator for latches and flip-flops. Some previous work in the literature are described and the results discussed. After, the model for the generator is described and it is proposed a way to improve the steady states coverage and the transition coverage. Finally it is described how it will be validated and a schedule.*

Resumo. *Este artigo descreve uma proposta para um trabalho de conclusão de curso. Neste artigo, é descrito um gerador de vetores de teste para latches e flip-flops. Comparamos com algums trabalhos anteriores existente na literatura e seus resultados são discutidos. Após, descrevemos a idéia por trás do gerador e é proposto uma maneira de melhorar a cobertura de estados e de transições. Finalmente, é descrito como será validado o trabalho e um cronograma.*

1. Introduction

The cell-based methodology for ASIC design, named Standard Cell, is very popular for its reduced project complexity, time-to-market and therefore overall cost. When using this technique, a more complex design is built from simpler logical blocks which are prevalidated, previous tested in order to see if the cell behavior is correct, and pre-characterized, when the timing and power characteristics are extracted taking into account the input slopes, temperature, power-supply voltage and etc. Furthermore, these blocks, also called cells, can be classified into 3 groups: inverters/buffers; combinational blocks and sequential/storage blocks. Usually, they are aggregated in a cell library. These libraries are made by Semiconductors fabrication companies, as known as, foundries which validate and characterize all the library's cells.

Realizing that a single library may be composed of a large number of cells and before characterization, it is important to validate the cells, a automated approach is necessary in order to speed up the process. The test of inverters can be simply done using a ring oscillator, and testing of combinational cells can be done using a combinational block where inside are the Circuit under Test (CUT), and a second stage which makes the output of the block equal to the input, as described at [3]. The test o sequential cell are more intricate to resolve because of the sequential nature of these blocks, *i.e.* the current output value is dependent of the current input and the past sequence of them. A solution was proposed using a shift-register for testing latches and an up-counter for flip-flops[5]. Another work proposed using a ring oscillator made of latches or flip-flops[4]. These works will be discussed further.

In this work, it is proposed a teat pattern generator for flip-flops and latches. The generator will cover all possible input states and theirs transitions. The generator will avoid repeated input transitions, be circular, *i.e.* begin and end in the same pattern and transition 1 input per step therefore easing debugging and avoid errors canceling each other out. Using the behavioral description of the cell and the test pattenr sequence, a template is created and compared with the CUT's output. If the coverage is less than 100%, multiples instantiations of the same cell may be used but with the inputs permuted or with one or more inputs negated. These multiple instances are chosen using the behavioral description and the sequence of inputs.

This article is organized as follows: In section 2, it is discussed the proposed work motivation; in section 3, previous works related to this one are analyzed and discussed; in section 4 it is described the generators project and a theoretical approach; in section 5 it is shown how the generator is going to be validated; and in section 6 a schedule for the project is shown.

2. Motivation

Today sequential logic is essential for the correct functioning of diverse systems being the the D-type flip-flop and D-type latch one of the most used sequential elements. This element is fundamental in order to build registers, which are used to in modern processors as a fast storage device and pipelines. Pipelines help parallelize multiple instructions avoiding idleness of the CPU and increasing the overall throughput of instructions.

Seeing the importance of sequential cells in today's circuits, testing them is very difficult. The main reason for it is that the current output state depends not only of the current inputs but the past input sequence, thus the problem may become time and memory consuming. Another problem is the presence of both set and reset signals which may cause a erasure effect, *i.e.*, the current state may not depends of the past input sequence anymore.

In addition, several sequential cells may compose a single Standard Cell Library and they require validation before being available for the customers, a automatic approach is necessary in order to reduce time-to-market and costs. Further more, some cells can have for a single logical function various implementation with different channel width and topology. A example of different D-type flip-flop topology is reported at [1]. The use of cells with same topology but with different transistor channel width is for timing, power and area constrains demanded by the target application. These factors increase the validation complexity of the library even more.

Taking these factors into account, a simple test generator is necessary in order to speed up the test phase. With a generator independent from the behavior, the test setup will be simpler and the test of several cells representing different logical functions can be made, increasing the test speed and the library validation. But, test coverage is also important factor to take in account and because of the sequential nature of these cells a high coverage is difficult to achieve.

3. Related Works

As mentioned in the introduction, some works in the literature show some approaches to test sequential cells, especially D-type latches with asynchronous set and reset (DLatSR) and D-type flip-flop with asynchronous set and reset (DFFSR).

At [4], it is proposed a ring oscillator made of 6 instantiation of the same DLatSR and another composed of 4 of the DFFSR. In each instance is represented a different output transition caused by different inputs transitions of the cell. Therefore, multiple input and output states and transitions are verified in each step of the ring. The figure 3 shows the test setup for latches and the figure 3 shows the setup for flip-flops. In both cases all the possible output transitions when an input is transitioned are covered because each of the ring step represent one input-output transition arc. However the coverage of steady states is not 100% for there are steady states that the ring can not reach and the coverage of a unexpected transition, *i.e.*, a input transition does not cause a transition in the output, are as well not totally covered. These cases must be evaluated because they are part of the expected behavior of the DLatSR and the DFFSR and might contain errors.

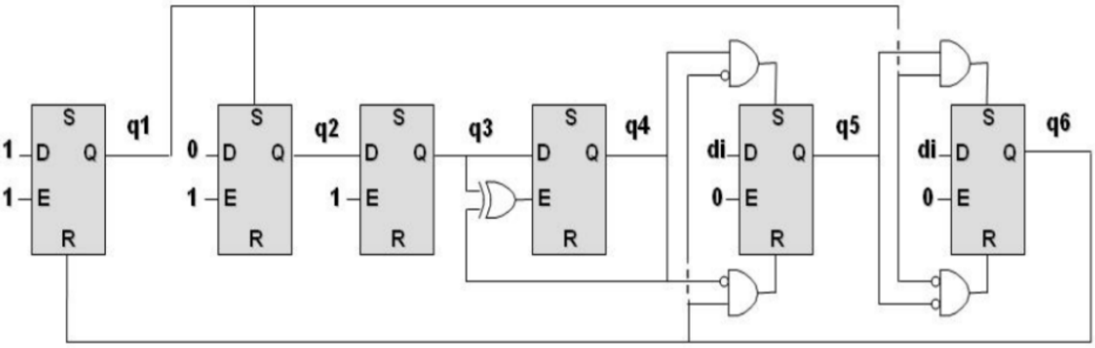


Figure 1. Ring Oscillator setup with latches described at [4]

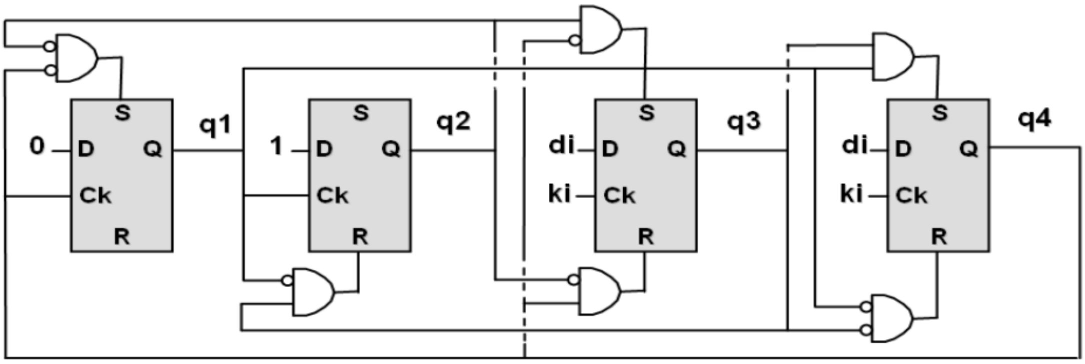


Figure 2. Ring Oscillator setup with flip-flops described at [4]

At [5], the testing of DLatSR is made by instantiating a 12 bit shift-register with the same cell as show by the figure 3. However some latches have theirs set and reset signal behavior determined by the current overall state of the register and the enable polarity signal is inverted at every couple of latches. The steady state coverage almost 100% but the state where both set and reset are on. However, some possible transitions and unexpected output transitions are not covered by this setup. For testing of DFFSR, a modified 5 bit counter is created where each bit is the same cell under test. The figure 3 shows the described setup. Similar as the test with the latch, the set and reset signals of several bits

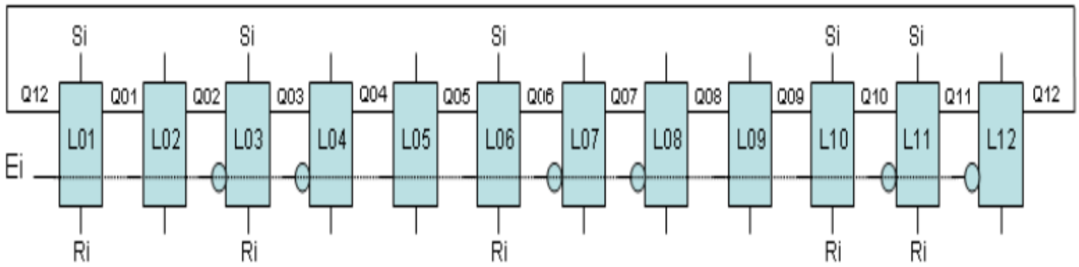


Figure 3. Ring Oscillator setup described at [5]

are dependent of the different state of the counter and are calculated by the handshake circuit. The test covers all steady states but the ones with both set and reset on and the transition coverage covers 50—5 of the unexpected transitions.

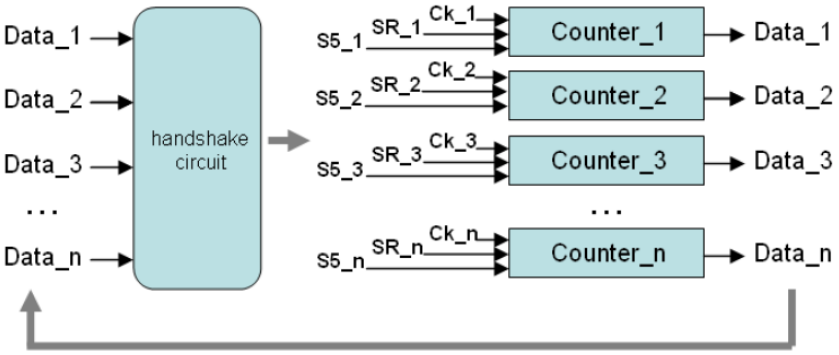


Figure 4. Modified counter described at [5]

Another approach is defined at [2], where a Finite State Machine is created using the cell behavior description. The FSM will pass through all states and transitions in order to create an input pattern sequence that has 100% coverage. Despite, being similar to our work, the article's generator is dependent of the cell behavior and ours is dependent only by the number of inputs. Another problem described is that the length of the sequence varies with the initial state.

4. Project Definition

The proposed generator will yield a group of signals which will be used as input by a Circuit under Test (CUT). In order to have a higher coverage, avoid error masking, it must conform to the following rules: (1) to generate all possibles states and transitions; (2) to be circular, *i.e.*, the initial state must be equal to the last state or there must be an 1-bit transition from the last state to the first one; (3) its transitions have a Hamming distance of 1, *i.e.*, the difference between two states is one bit, *e.g.* $0010 \rightarrow 0110$.

Using this set of rules, the generator can be modeled as an n-cube or an n-hypercube where each vertex receives a input state and each one of its neighbors has a state which differs by 1 bit. The edges represent a valid transition between two states.

Since it is interesting to cover both rising transition, when a bit goes from logic value ”0” to value ”1”, and falling transition, when a bit goes from logic value ”1” to value ”0”, the hypercube is bi-directed at every connection between two vertex. Fig. 5 shows the modeled graph. This graph has 2^N nodes and $N * 2^N$ edges, being N the number of inputs in the CUT.

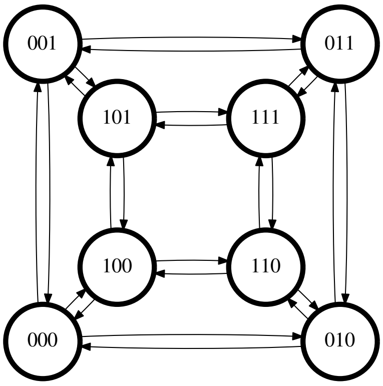


Figure 5. 3-Dimensional Hypercube representing a 3 bit Gray Code

In order to cover all transitions it is necessary to find out an Euler cycle, *i.e.*, a cycle that passes through all edges exactly once beginning and ending at the same node. Due to the form of the graph, it has many different Euler cycles. In order to find a single cycle, some of the algorithms present in the literature can be used. However, some properties of the modeled graph and the given problem can be used to find out a simpler solution to the problem.

Since each node is labeled with a state which can be interpreted as binary numbers, they can be ordered using the natural number ordering. Then, they can also be organized in a tree structure with the smallest state, the one with all the inputs at the logic state ”0”, as the root. The state with all inputs at ”0” is used as root because it is the smallest integer. Then, all nodes are placed in a way that theirs sons must have a label whose value is grater than the father. But in order to represent all possible 1-bit transitions depicted in the original graph, some nodes must be repeated and placed as sons of each of its smaller label neighbors. This happens because one node can have several smaller neighbors. Fig 6 shows the results of the tree transformation on the graph depicted in Fig 5. With the tree built, it is only necessary to make a search through it and the state sequence is created.

In order to compare the cells output and verified their behavior, a template must be created using the cells behavior and the generator sequence. It will also conform to a set of rules. It will change its output with the generator and output the desired result that will be compared with the output of the CUT. As discussed previously, the sequential cells have a memory effect, a input state may have more than one output state and they must be tested. Since the generator is behavior independent and in order to cover as many states and transitions as possible, one strategy possible is the use of multiple instantiation of a cell but with the inputs permuted or one or more input bits negated. This means that in parallel, the generator runs through multiples Euler cycles at the same time.

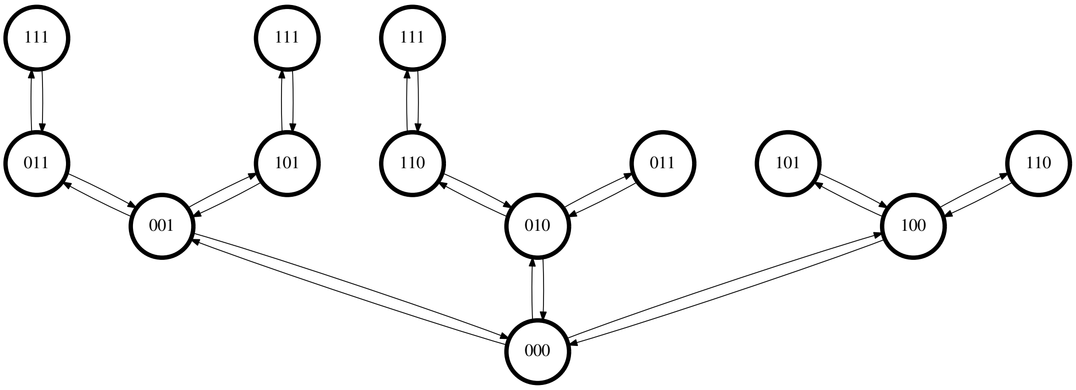


Figure 6. The resulting tree from the graph in Fig 5

5. Validation

In order to validate this work, a application program will be made in order to verify the steady states and transitions coverage. The program will have a component describing the cell's behavior and will output the steady states and the transition. The program will calculate a sequence and calculate the steady coverage and transition coverage. With this information, the program can choose which instantiation of the same cell , as described in section 4, will be used in order to increase both steady and transitions coverage.

Further validation can be made integrating the previous program with the SPICE simulator. The program will calculate the input pattern sequence, the template containing the expected result and how many cell must be instantiated. After, a SPICE script will be automatically created containing the generator, the cells and the template calculated by the previous software.

The group of cells that will be used to validate this work is composed of D-type latches, D-type flip-flops and T-type flip-flops.

6. Schedule

- First month:
    - Language choice
    - Setting of the data structure
    - Implementation
    - Final program debugging and fixes if necessary
    - Preliminary results analysis
- Second and Third month:
    - SPICE language syntax learning
    - SPICE-application integration implementation
    - Result analysis
- Forth and Fifth month:
    - Undergraduate thesis writing

References

[1] Massimo Alioto, Elio Consoli, and Gaetano Palumbo. Analysis and comparison in the energy-delay- area domain of nanometer cmos flip-flops: Part i—methodology and design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, June 2011.

[2] Helder H. Avelar, Paulo F. Butzen, and Renato P. Ribas. Automatic circuit generation for sequential logic debug. *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2015.

[3] Renato P. Ribas, Vinicius Callegaro, Marcelo Lubaszewski, André Ivanov, and André I. Reis. Circuit design for testing standard cell libraries. *WCAS 2011, Workshop on Circuits and System Design*, 1, 2011.

[4] Renato P. Ribas, Yuyang Sun, André I. Reis, and André Ivanov. Ring oscillators for functional and delay test of latches and flip-flops. *SBCCI '11 Proceedings of the 24th symposium on Integrated circuits and systems design*, 2011.

[5] Renato P. Ribas, Yuyang Sun, André I. Reis, and André Ivanov. Self-checking test circuits for latches and flip-flops. *On-Line Testing Symposium (IOLTS), IEEE 17th International*, 2011.