UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

FELIPE DOS SANTOS MARRANGHELLO

Logic synthesis for sequential material implication logic based on resistance switching devices

Thesis presented in partial fulfillment of the requirements for the degree of Doctor in Microelectronics

Advisor: Prof. Dr. Renato Perez Ribas
Co-advisor: Prof. Dr. André Inácio Reis

Porto Alegre
2017

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Marranghello, Felipe dos Santos

Logic synthesis for sequential material implication logic based on resistance switching devices / Felipe dos Santos Marranghello. – 2017.

120 f.:il.

Orientador: Renato Perez Ribas; Co-orientador: André Inácio Reis.

Tese (Doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2017.

1.Síntese lógica. 2.Implicação material 3.Dispositivos memristivos. I. Ribas, Renato Perez. II. Reis, André Inácio. III. Logic synthesis for sequential material implication logic based on resistance switching devices.

AGRADECIMENTOS

Agradeço ao meu pai Carlos Roberto Marranghello, à minha mãe Eunice dos Santos Marranghello e à minha irmã Letícia dos Santos Marranghello.

Agradeço aos meus orientadores Renato Perez Ribas e André Inacio Reis. Aos colegas de laboratório e a todos que me apoiaram durante o caminho.

Agradeço ao CNPq pelo financiamento durante o doutorado.

Síntese lógica para lógica de implicação sequencial usando dispositivos com resistência variável

Dispositivos de resistência variável (RSD) são alternativas promissoras para a criação de memórias não voláteis (NVM). Estas memórias também podem influenciar o projeto de circuitos digitais através de “lógica em memória”. Dentre tais paradigmas lógicos está a lógica de implicação material (RSD-IMP). A principal diferença entre RSD-IMP e lógica convencional é que em RSD-IMP as funções Booleanas são computadas através de uma sequência de operações de implicação material (IMP). Tais operações também são chamadas de instruções. Neste sentido, o paralelismo de circuitos digitais convencionais não é observado em RSD-IMP porque uma única instrução é feita por ciclo. Esta tese propõe métodos de síntese lógica para RSD-IMP. Dada uma representação de uma função Booleana, o objetivo é obter uma sequência de operações em RSD-IMP que corresponda a esta função. As métricas para avaliar a qualidade de uma solução são o número de instruções e o número de RSD. Um ponto interessante de RSD-IMP é que, para qualquer função Booleana de n variáveis, existe uma sequência de instruções para esta função que necessita de apenas n+2 RSD. A principal maneira de se obter tal sequência é através de uma forma Booleana recursiva (RBF) correspondente à função alvo. A primeira contribuição deste trabalho é a proposta de um método mais eficiente para sintetizar RBF a partir de soma-de-produtos (SOP). Então, o conceito de RBF é generalizado para soma-de-RBF (SRBF). É demonstrado que SRBF também podem ser diretamente transformadas em uma sequência de instruções que pode ser computada com n+2 RSD. Relaxando a restrição de n+2 RSD para n+k RSD, com $k \geq 2$, é possível explorar a classe de RBF fatorada (FRSBR). Finalmente, é discutido o projeto lógico de somadores binários baseados em RSD-IMP.

Palavras-chave: Síntese lógica, lógica de implicação material, memristors, dispositivos com resistância variável, circuitos digitais, funções Booleanas.

Logic synthesis for sequential material implication logic based on resistance switching devices

Resistance switch devices (RSD) are promising alternatives to implement nonvolatile memories (NVM). These memories can also influence the design of digital circuits through logic-in-memory. Among these novel logic paradigms is the material implication (RSD-IMP) logic. The main difference between RSD-IMP logic from conventional digital circuit design is that Boolean functions are evaluated in RSD-IMP logic as a sequence of material implication (IMP) operations, known as instructions. In this sense, the parallelism observed in standard digital design is not obtained because a single IMP operation is performed per cycle. This thesis focuses on logic synthesis methods for RSD-IMP logic. Given a standard description of a Boolean function, the goal is to obtain a sequence of operations in RSD-IMP logic to evaluate the target function. The standard cost metrics are the number of instructions and the number of RSD required. An interesting aspect of RSD-IMP logic is that, for any n-input Boolean function f, there is a sequence of instructions in RSD-IMP that evaluates f using n+2 RSD. The main method to obtain such a sequence of instructions is to synthesize a recursive Boolean form (RBF) for f. The first contribution of this thesis is a more efficient method to synthesize RBF from a sum-of-products (SOP). Moreover, the concept of RBF is generalized to obtain a broader class of expressions that can be transformed into sequence of operations requiring only n+2 RSD. This new class of expressions is named sum-of-RBF (SRBF). Furthermore, the constraint of n+2 RSD is relaxed to allow n+k RSD, where $k \geq 2$ is an arbitrary integer. By relaxing this constraint, the class of factored SRBF (FSRBF) is obtained. The number of additional RSD can be controlled by considering the logic depth of FSRBF during the logic synthesis process. Finally, the logic design of binary adders in RSD-IMP logic is discussed.

Keywords: Logic synthesis, material implication logic, memristors, resistance switching devices, digital circuits, Boolean functions.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-inverter-graph
AP	Antiparallel
BDD	Binary decision diagram
FA	Full-adder
FL	Free layer
FLF	Four-level-form
FSRBF	Factored sum-of-recursive-Boolean-forms
IMP	Material implication
ISOP	Irredundant sum of products
LUT	Look-up table
MTJ	Magnetic tunnel junction
POS	Product-of-sums
PP	Parallel
RBF	Recursive Boolean Form
RCA	Ripple-carry adder
RL	Reference layer
RSD	Resistance switching device
RSD-IMP	Material implication logic based on RSD
SC-FSRBF	Single-cube factored recursive Boolean forms
SCE	Short circuit evaluation
SOP	Sum of products
SRBF	Sum of Recursive Boolean Forms
STT	Spin-transfer-torque
VO	Oxygen vacancy

SUMMARY

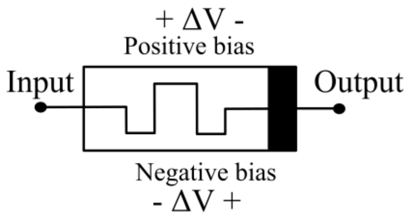
1. INTRODUCTION

Advances on nanotechnologies processes have enabled the development of novel beyond CMOS devices that may be exploited in future computing systems. Among such devices, resistance switching devices (RSD), also known as memristors or memristive devices (STRUJOV, 2008), (CHUA, 2011), have great potential to be exploited in nonvolatile memories (NVM) (AKINAGA, 2010), (KENT, 2015). Moreover, RSD can also lead to novel alternatives to perform logic. In particular, RSD is suitable to logic-in-memory paradigms where logic is performed directly at the memory bits without the need to transfer data to a processing unit (ZHU, 2013), (WONG, 2015). In this sense, logic-in-memory may contribute to reduce the memory bottleneck.

RSD have been investigated since the 1960's (HICKMOTT, 1962), (GIBBSONS,1964), (SIMMONS, 1967). The state of a RSD can be toggled between a low-resistance state (LRS) and a high-resistance state (HRS). Therefore, the resistance state can be used to encode binary information, where the LRS represents the logic level '1' whereas the HRS represents the logic level '0'.

The basic structure of a RSD is a capacitor like metal-insulator-metal (MIM) structure, (KENT, 2015), (MEENA, 2014), (PAN, 2014), (VALOV, 2013), (WONG, 2012), (WOUTERS, 2015). However, by applying a voltage bias, the insulator can be modified to become a conductor. The transition between the insulator and conductor states corresponds to the transitions between the HRS and LRS. This transformation can occur due to different physical phenomena, including the creation/rupture of a conductive filament connecting the metallic terminals and a phase change from a highly resistive amorphous phase to a conductive crystalline phase. Since these transformations require a minimum electrical field to occur, it is possible to apply small voltage bias without modifying the device state. Fig. 1.1 illustrates the electrical symbol of a RSD.

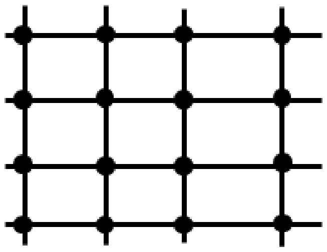
Figure 1.1 – Symbol of a RSD. The output terminal is marked by a thick black line. A positive voltage bias reduces the resistance value, whereas a negative voltage bias increases the resistance value.



Source: The author.

The main application of RSD is on the implementation of NVM. The basic structure of a NVM based on RSD is a matrix comprising horizontal and vertical wires, also known as crossbar. The basic structure, known as passive crossbar, is illustrated in Fig. 1.2 (MEENA, 2014). The sets of vertical and horizontal wires are at different plans. At the intersection between a row and a column, there is a RSD connecting the wires. In Fig. 1.2, the black dots represent the RSD. The passive crossbar structure is interesting due to the possible high density. Each bit requires an area of $4L^2$, where L is the minimum feature size.

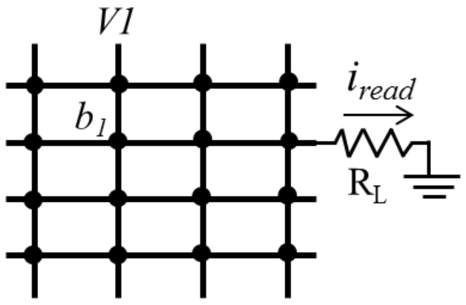
Figure 1.2 – Basic crossbar structure, a RSD is placed at each intersection.



Source: (MEENA, 2014).

A RSD within the crossbar can be accessed by applying a voltage bias between the row and column where the device is placed and measuring the current through the RSD. The process is illustrated in Fig. 1.3, where the state of device b_1 is read by applying a voltage bias *V1* to the device. The state of the device can be defined by sensing either the current or a voltage across a load resistor R_L. If current i_{read} across R_L is measured, then a high value of i_{read} indicates that b_1 is in the LRS. Conversely, a low value of i_{read} means that b_1 is in the HRS.

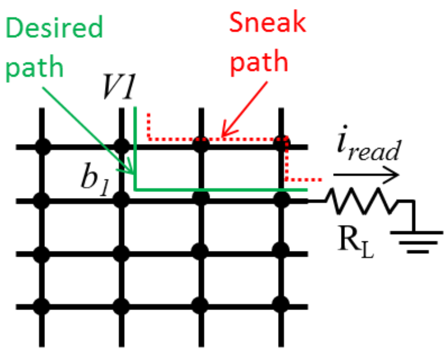
Figure 1.3 – Basic reading scheme in a RSD crossbar.



SOURCE: The author.

The main problem with this approach is the existence of sneak paths. A sneak path is an undesirable path for the current, as illustrated in Fig. 1.4. In Fig. 1.4, the solid green line represents the desired path for the current whereas the dotted red line represents one of many possible sneak paths. Sneak paths can make i_{read} to increase such that a device in the HRS is wrongly identified as being in the LRS. The sneak path current can be reduced by connecting the RSD to a selector which can be either a transistor or a diode, by using two RSD in an anti-serial connection or by applying an intermediate voltage bias to unselected lines and rows. It has been shown that a single RSD can behave as a RSD connected to a diode or as two RSD in anti-serial connection.

Figure 1.4 – Sneak path problem in RSD crossbar.



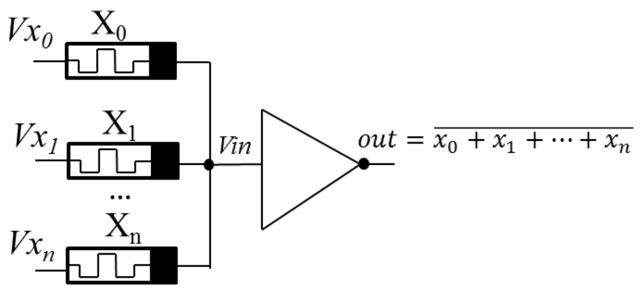
SOURCE: The author.

RSDs have also been used in other applications in addition to NVM such as the design of field-programmable gate array architectures (FPGA) (GUO, 2017), neuromorphic systems (INDIVERI, 2015) and digital integrated circuit design. In the following, we focus on applications of RSD in digital design.

1.1 LOGIC STYLES FOR RESISTANCE SWITCHING DEVICES

RSD can be exploited in several manners to perform logic. RSD can be combined with CMOS transistors to create high fanin NOR and NAND gates, as illustrated in Fig. 1.5 (KVATINSKY, 2012). In order to understand the behavior of the circuit, let all $V_{x_i} = 0$ and all X_i be in the HRS. If V_{x_0} rises to the power supply voltage (*Vdd*), then voltage *Vin* remains close to 0 until X_0 switches to the LRS. After this point, *Vin* rises to *Vdd*. As *Vin* rises, the remaining devices become negatively biased, as illustrated in Fig. 1.1 and enter the HRS. The resulting voltage divider allows *Vin* to reach a value near close to *Vdd*. The reverse current through the other RSD can be further reduced by using self-rectifying RSD.

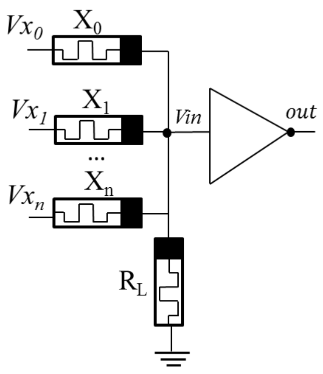
Figure 1.5 – High fanin NOR gate using a hybrid RSD/CMOS structure.



Source: (KVATINSKY, 2012).

Hybrid RSD/CMOS gates can also be used to implement threshold logic (GAO, 2013), (FAN, 2014), (JAMES, 2014). One example of a hybrid RSD/CMOS gate is shown in Fig. 1.6 (JAMES, 2014). The basic structure is similar to the one shown in Fig. 1.5 with the inclusion of a load RSD that represents the threshold value of the function. The basic idea of this approach is that voltage *Vin* only rises to *Vdd* if there are enough inputs in *Vdd.* In order to fully benefit from these novel gates, logic synthesis methods focusing on threshold logic are important (NEUTZLING, 2014), (PALANISWAMY, 2014), (NEUTZLING, 2015), (KULKARNI, 2016). Notice that such hybrid approaches are increments on standard cell based design and are not suitable for logic-in-memory applications.

Figure 1.6 – Hybrid RSD/CMOS threshold logic gate.

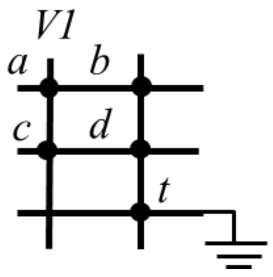


Source: (JAMES, 2014).

Logic circuit topologies for logic-in-memory using RSD can be broadly classified into switch based, instruction based and hybrid. In switch based approaches, the memory is configured as a switch network (LEEVY, 2014), (ALAMGIR, 2016), (PAPANDROULIDAKIS, 2017). A voltage is applied to a certain row and a target RSD is driven to the LRS only if the function represented by the given structure evaluates to 1. Fig. 1.7 shows an example of a crossbar structure configured as switch network, where RSD are placed in the intersections between the rows and columns. The missing RSD is fixed to the HRS. Voltage *V1* only propagates to RSD *t* when the following function *f* evaluates to true:

$$f = ab + cd \qquad (1.1.1)$$

Figure 1.7 – Switch network like implementation for $f = ab + cd$ in RSD crossbar (the missing RSD is tied to the HRS).

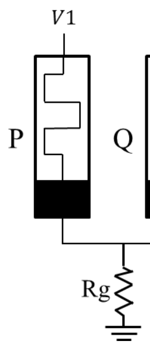


Source: The author.

Switch based methods can benefit from standard algorithms for switch network generation (POSSANI, 2016), (KAGARIS, 2016), even though modifications may be required in order to consider the topology constraints of RSD-NVM. Notice that, even though the evaluation of the functions requires a single cycle, the time required to configure the crossbar in the desirable manner must be accounted for.

In instruction based approaches, the computation is performed as a sequence of basic instructions. The basic instruction can be either the material implication (IMP) (BORGHETTI, 2010), (ADAM, 2016), which is referred to as RSD-IMP, or the 3-input majority function (LINN, 2012), (GAILLARDON, 2016). The sequential behavior of these approaches represents a major difference to usual digital circuit design. Therefore, exploiting standard logic synthesis algorithms is not straightforward. This has motivated the development of novel logic synthesis algorithms focusing both majority logic (SHIRINZADEH, 2016), (SOEKEN, 2016) and RSD-IMP logic (LEHTONEN, 2010), (MARRANGHELLO, 2015a), (MARRANGHELLO, 2015b), (POIKONEN, 2012), (RAGHUVANSHI, 2014), (TEODOROVIC, 2013). The basic idea in IMP logic is to consider a structure as shown in Fig. 1.8(a), where two RSD P and Q are connected to a load resistor Rg through a common node. By applying appropriate voltages V1 and V2 to P and Q, respectively, the final state of Q, denoted by *q'*, becomes a function of the state of P, denoted by p. More specifically, the final state of Q is the LRS if the initial state of Q is the LRS or if P is in the HRS. This behavior can be translated into an IMP operation (BORGHETTI, 2010), as shown in Fig. 1.8(b). In Chapter 3, the behavior of IMP logic is detailed.

Figure 1.8 – Basic structure for RSD-IMP logic: (a) basic structure and (b) corresponding truth table.



p	*q*	*q'*
0	0	1
0	1	1
1	0	0
1	1	1

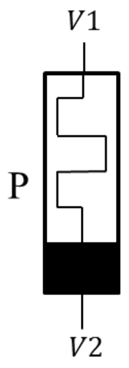
(a) (b)

Source: (BORGHETTI, 2010).

The idea for the RSD majority logic is to consider a single RSD P with voltages *V1* and *V2* applied to its terminal, as shown in Fig. 1.9(a). Let p and *p'* denote the initial and final states of P, respectively. Fig. 1.9(b) shows the value of *p'* as function of p, *V1* and *V2*. The logic behavior corresponds to the following expression: (LINN, 2012):

$$p' = \overline{V2}(p + V1) + pV1 \qquad (1.1.1)$$

Figure 1.9 – RSD based majority logic: (a) basic structure and (b) corresponding truth table.



p	*1*	*V2*	*p'*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

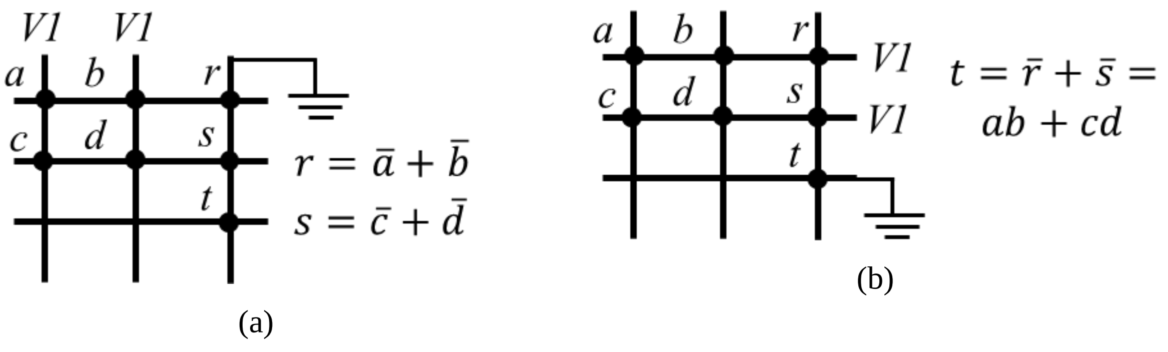
(a) (b)

Source: (LINN, 2012).

Hybrid approaches configure the crossbar as a switch network but control the data flow through instructions (KVATINSKY, 2014), (XIE, 2017). Fig. 1.10 presents an example of such hybrid methods based on the work described in (KVATINSKY, 2014), where *a*, *b*, *c* and *d* are the inputs while *r*, *s* and *t* are auxiliary RSD. Missing RSD are tied to the HRS. By applying an adequate voltage *V1* to the columns where the inputs are placed, *r* can be programmed to store $\bar{a} + \bar{b}$ while *s* stores $\bar{c} + \bar{d}$, as shown in Fig. 1.10(a). Then, in a second step, *t* is programed to store $ab + cd$, as shown in Fig 1.10(b). Notice that the topology shown in Fig. 1.10 is similar to the shown in Fig. 1.8. The difference is the utilization of auxiliary RSD *r* and *s* to store intermediate values.

Overall, all approaches suffer from the need of a control block to either configure the crossbar, as in the case of switch based approaches, or to provide the correct control signals, as in the case of instruction based approaches. Hybrid approaches may fit into both cases. In this sense, reducing the number of RSD and the number of instructions should also reduce the size of the control block.

Figure 1.10 – Example of a hybrid approach for logic-in-memory using RSD: (a) first cycle to compute intermediate values, and (b) final cycle to evaluate the final value of the function.



Source: The author.

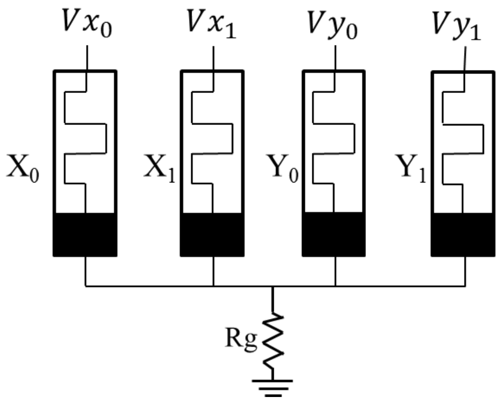
This thesis focuses on RSD-IMP logic. RSD-IMP appears to provide a good trade-off between the number of cycles and the number of RSD. However, it is not the goal to determine the best approach. It is worth noticing that, even though we have the RSD-IMP logic as motivation, this thesis mostly discusses synthesis and optimizations of different types of Boolean expressions. In this sense, the discussion presented herein may be useful to different logic topologies.

1.2 OVERVIEW OF RSD-IMP LOGIC

This Section presents an overview of RSD-IMP logic. A more detailed explanation is presented in Section 3.2. In RSD-IMP logic, binary values are represented through the resistance of a RSD. A RSD in the LRS represents the logic value ‘1’, whereas a RSD in the HRS represents the logic value ‘0’.

The basic structure of RSD-IMP logic comprises several RSD connected to a common node and to a load resistor Rg. There are n RSD that store the primary input variables and $k \geq 2$ auxiliary RSD. These auxiliary RSD correspond to auxiliary variables and are used to perform computation. Fig. 1.11 presents a simple example of a RSD-IMP block with two input variables and two auxiliary RSD. Computation is performed as a sequence of instructions. Each instruction is obtained by applying a voltage bias to different RSD. Due to the common node, the voltage bias across a RSD depends on the bias applied to others RSD. This dependence allows logic to be performed.

Figure 1.11 – Basic RSD-IMP logic block with two inputs and two auxiliary RSD.



Source: The author.

When performing an operation, each RSD can either be set to a high-impedance state or be biased by one of the three different control voltages. These three control voltages are V_{OFF}, V_{ON} and V_{COND}. V_{OFF} corresponds to a voltage value that forces the RSD to the HRS, V_{ON} sets the RSD to the LRS and V_{COND} does not change the state of the RSD.

The operations that can be performed in RSD-IMP logic are the reset, single-input implication and multi-input implication. The reset operation sets one or more RSD to the HRS. The reset operation is performed by applying V_{OFF} to the target RSD.

A single-input implication between two variables x_0 and y_0 ($x_0 \rightarrow y_0$) is performed by applying V_{COND} to X$_0$ and V_{ON} to Y$_0$. If $x_0 = 0$, then V_{ON} sets Y$_0$ to the LRS and $y_0 = 1$.

Otherwise, $x_0 = 1$, V_{COND} reduces the voltage across Y$_0$ such that V_{ON} does not suffice to cause a transition from the HRS to the LRS. In this case, the value *y$_0$* is not changed. Therefore, the final value of *y$_0$* is 1 if the initial value of *y$_0$* is 1 or if x_0 is equal to 0. This behavior can be interpreted as the material implication operation (IMP), as follows:

$$y_0' \coloneqq x_0 \rightarrow y_0 = \overline{x_0} + y_0 \tag{1.2.1}$$

where y_0' is the next state of y_0. Since the result of the IMP operation is always stored at y_0, we write (1.2.1) simply as $x_0 \rightarrow y_0$.

To perform a multi-input implication, V_{COND} is applied to several devices X$_1$,…, X$_n$ while V_{ON} is applied to a target device Y$_0$ (SHIN, 2011). If all devices to which V_{COND} is applied are in the HRS, then the target device switches to the LRS. Otherwise, the state of the target device does not change. Therefore, the final value of y_0 is 1 if the initial value of y_0 is 1 or if all $x_1, \ldots, x_n$ are equal to 0. A multi-input implication can be written as follows:

$$(x_1 + \cdots + x_n) \rightarrow y_0 = \overline{x_1} \ldots \overline{x_n} + y_0 \tag{1.2.2}$$

Notice that the correct behavior of the circuit depends on defining adequate values for V_{OFF}, V_{ON}, V_{COND} and Rg. Several works have proposed methodologies to define such values. In Section 3.2, we detail how these values can be defined.

Example 1.2.1: A sequence of operations for the 2-input AND (AND2), x_0x_1, using two auxiliary variables y_0 and y_1, is given in Table 1.2.1.

Table 1.2.1 – Evaluation of AND2 in RSD-IMP logic.

	Instruction	Result
1.	$y_0 = 0, y_1 = 0$	
2.	$x_0 \rightarrow y_0$	$y_0 = \overline{x_0}$
3.	$x_1 \rightarrow y_0$	$y_0 = \overline{x_0} + \overline{x_1}$
4.	$y_0 \rightarrow y_1$	$y_1 = \overline{\overline{x_0} + \overline{x_1}} = x_0x_1$

Source: (BORGHETTI, 2010).

1.3 CHALLENGES AND MOTIVATION

In RSD-IMP logic, there can have different sequence of instructions for the same target function. The quality of the solution can be measured in terms of number of instructions and number of RSDs required. For instance, the solution for AND2 operation presented in Table 1.2.1 takes four instructions and four RSDs.

Example 1.2.2: In this example, we compare different solutions for the 2-input OR (OR2), $x_0 + x_1$. The first solution, shown in Table 1.2.2, comprises six cycles and uses five RSDs. By adding a RSD y_2 , the reset operation in cycle 4 can be skipped. The resulting sequence of operations becomes as shown in Table 1.2.3.

Table 1.2.2 – Evaluation of OR2 in six cycles using four RSDs.

	Instruction	Result
1.	$y_0 = 0, y_1 = 0$	
2.	$x_0 \rightarrow y_0$	$y_0 = \overline{x_0}$
3.	$y_0 \rightarrow y_1$	$y_1 = x_0$
4.	$y_0 = 0$	$y_0 = 0$
5.	$x_1 \rightarrow y_0$	$y_0 = x_1$
6.	$y_0 \rightarrow y_1$	$y_1 = x_0 + x_1$

Source: The author.

Table 1.2.3 – Evaluation of OR2 in five cycles using five RSDs.

	Instruction	Result
1.	$y_0 = 0, y_1 = 0, y_2 = 0$	
2.	$x_0 \rightarrow y_0$	$y_0 = \overline{x_0}$
3.	$y_0 \rightarrow y_1$	$y_1 = x_0$
5.	$x_1 \rightarrow y_2$	$y_2 = x_1$
6.	$y_2 \rightarrow y_1$	$y_1 = x_0 + x_1$

Source: The author.

Both solutions, shown in Table 1.2.2 and in Table 1.2.3, use only reset and single-input operations. The solution can be improved by using multi-input implications, as shown in Table 1.2.4. The solution shown in Table 1.2.4 takes only three cycles and uses four RSDs.

Table 1.2.4 – Evaluation of OR2 in three cycles using four RSD.

	Instruction	Result
1.	$y_0 = 0, y_1 = 0$	
2.	$(x_0 + x_1) \rightarrow y_0$	$y_0 = \overline{x_0}\,\overline{x_1}$
3.	$y_0 \rightarrow y_1$	$y_1 = x_0 + x_1$

Source: The author.

It has been shown that for any n-input Boolean function, there is a sequence of operations that requires $n + 2$ RSD (LEHTONEN, 2010). In this sense, three logic synthesis challenges, regarding RSD-IMP logic, are the following:

1) Find the smallest sequence of operations that can be computed with $n + 2$ RSD.
2) Find the smallest sequence of operations that require at most $n + k$ RSD, where $k \geq 2$ is an arbitrary value.
3) Find the smallest sequence of operations with no upper bound on the number of RSD used.

Most works in the literature targeting RSD-IMP logic focuses on the first challenge (POIKONEN, 2012), (TEODOROVIC, 2013), (RAGHUVANSHI, 2014). In (LEHTONEN, 2010), it is proposed the use of recursive Boolean forms (RBF). A RBF can be defined as follows:

$$f = f_0 + \overline{(f_1 + \overline{(f_2 + \overline{\ldots + \overline{(f_{\phi-2} + \overline{f_{\phi-1}})}})})} \tag{1.2.3}$$

where each f_i is a negative unate SOP and ϕ is the number of levels in f. The interest on RBF arises because such a kind of forms can always be directly translated to a sequence of instructions that can be evaluated with $n + 2$ RSD. Moreover, each cube in the RBF corresponds to an instruction. Therefore, methods to optimize RBF have been proposed. The methods described in (RAGHUVANSHI, 2014) and in (POIKONEN, 2012) are based on finding a cover for the function through Karnaugh maps. In (TEODOROVIC, 2013), a graph based method, where each vertex is a minterm, is presented. Such a method is similar to the cover based methods. All these approaches work over representations that always use 2^n elements for an n-input Boolean function, being restricted to somewhat simple functions. In (WANG, 2016), a genetic algorithm to optimize RBF is proposed.

The works discussed in (CHAKRABORTI, 2014) and in (CHATTOPADHYAY, 2011), propose a BDD-based and an AIG-based method, respectively. In both cases, each node of the graph is directly transformed into a sequence of instructions. In order to evaluate only once the nodes with fanout greater than one, more than $n + 2$ RSD are used.

1.4 THESIS PROPOSAL

This thesis focuses on defining and synthesizing classes of Boolean expressions such that: (1) the size of the expression is directly related to the number of instructions; and (2) the number of required RSD can be derived in linear time with respect to the size of the expression. From such expressions, we proposes logic synthesis methods for RSD-IMP logic considering the challenges previously described.

We consider RBF as the base form and then generalize RBF to other forms. For each new form, we investigate how this form can be transformed into a sequence of instructions, as well as the resulting number of instructions and RSD. Then, we derive algorithms to synthesize such forms while optimizing the resulting sequence of instructions.

We begin by developing algorithms related to the first challenge, *i.e.*, to find the smallest sequence of operations that can be computed with $n + 2$ RSDs. Our first contribution is related to the synthesis of RBF. In order to improve the scalability of RBF synthesis methods, we develop a RBF synthesis method that can be applied over different representation such as sum-of-products and BDD. Therefore, we discuss how RBF can be optimized by removing redundant cubes.

Our second contribution related to the first challenge is the proposal of sum-of-RBF (SRBF). By generalizing RBF into SRBF, we are able to reduce the number of instructions while respecting the lower bound of $n + 2$ RSDs. We also propose a SOP-based algorithm for the synthesis of SRBF.

In order to exploit an arbitrary number $n + k$ of RSDs, where $k \geq 2$, related to the second challenge (*i.e.*, to find the smallest sequence of operations that require at most $n + k$ RSD), we propose the use of factored SRBF (FSRBF). More specifically, we focus on single-cube FSRBF (SC-FSBRF). We show that the number of levels in the SC-FSRBF is directly related to the number of RSDs required to evaluate the resulting sequence of operations. The SOP-based algorithm for SRBF synthesis is expanded to SC-FSRBF.

Regarding the third challenge, i.e., to find the smallest sequence of operations with no upper bound on the number of RSDs applied, we propose a SOP-based approach to minimize the sequence of instructions when there is no maximum bound in the number of RSDs. This last approach takes into account the benefits of having all variables in both direct and complementary forms while considers the extra cost to obtain the complement of a variable. This problem has been addressed in (XIE, 2017), which evaluates any Boolean function in seven cycles. In this sense, we obtain a different trade-off between the number of RSDs and the number of instructions.

We finish our contributions by discussing the logic design of binary adders. In contrast to previous contributions, the design of binary adders takes into account the matrix structure of the RSD memory. The sequence of instructions to implement a full-adder (FA) circuit is obtained from a SRBF. Thus, we explore adders designs based on the proposed FA.

1.5 TEXT ORGANIZATION

In Chapter 2, we provide a background on logic synthesis field. We discuss terms and definitions useful for the overall understanding of the thesis.

Chapter 3 focuses on RSD and RSD-IMP logic. Section 3.1 describes the basic physical behavior of RSD. Section 3.2 details the behavior of RSD-IMP logic, and Section 3.3 discusses existing logic synthesis methods for RSD-IMP logic.

Chapter 4 focuses on recursive Boolean forms (RBF). RBFs are the most studied forms for RSD-IMP because these ones can always be translated into a sequence of instructions computable with $n + 2$ RSDs in linear time with respect to the size of the RBF. We propose a more efficient method to evaluate RBF as well as two algorithms to synthesize RBF. The first algorithm provides optimal RBF that is well suited for simple functions with at most four inputs. The second algorithm aims to handle more complex functions at the cost of suboptimal solutions. The proposed methods show significant improvements over existing approaches.

In Chapter 5, we propose the concept of sum-of-RBF (SRBF). We demonstrate that SRBF can be transformed into a sequence of instructions that requires $n + 2$ RSDs. In Section 5.1, we present a SOP based algorithm to synthesize SRBF. In Section 5.2 we propose the use of factored SRBF (FSRBF) which can benefit from more than $n + 2$ RSD.

In Chapter 6, we present two other contributions. In Section 6.1, we evaluate the benefits of having the variables available in both polarities as a method to reduce the number of cycles (instructions) in RSD-IMP logic. In Section 6.2, we propose a novel logic design for a binary adder in RSD-IMP logic. The basic full-adder block is obtained from the developed methods. Then, we propose new implementations of binary adders that take into account different trade-offs between the number of instructions and the number of RSDs. Finally, Chapter 7 presents the conclusions and future works.

2 LOGIC SYNTHESIS BACKGROUND

This chapter presents some fundamentals on different concepts that are related to this work. Some helpful references for such a background are (BRAYTON, 1982), (BRAYTON, 1984) and (DE MICHELLI, 1994).

2.1 DEFINITIONS

An *n*-input Boolean function *F*(X) defined over the variable set X with $m \geq 1$ outputs is a relation $F(X) = \{0,1\}^n \mapsto \{0,1,-\}^{m,}$ where '$-$' denotes *don't care*. When $m = 1$, the function is a single output function. For m larger than 1, the function corresponds to a multi-output function. Constant functions false and true are denoted by 0 and 1, respectively. The on-set of *F* is denoted by F_{ON} and consists of all input assignments **x** such that $F(\boldsymbol{x}) = 1$. Similarly, the off-set of *F* (F_{OFF}) and the *don't care* set of *F* (F_{DC}) are all input assignments **x** for which $F(\boldsymbol{x}) = 0$ and $F(\boldsymbol{x}) = -$, respectively. If $F_{DC} = \{\}$, *F* corresponds to a completely specified function (CSF). Otherwise, *F* is an incompletely specified function (ISF). A given function *F* contains (or dominates) a function G if $(G_{ON} \cup G_{DC}) \subseteq (F_{ON} \cup F_{DC})$.

A cofactor of *F* with respect to a variable x_i is a function obtained by assigning x_i to 1 or 0 in F. If $x_i = 0$, we obtain the negative cofactor. If $x_i = 1$, we obtain the positive cofactor. If the negative and positive cofactors of F with respect to x_i are equal, then x_i corresponds to a *don't care* variable. In other words, the value of F does not depend on x_i. The support of a given function *F* is the set of variables that are not *don't care* in F.

A Boolean function *F* is said to be positive unate on a variable x_i if $F(x_1,\ldots,1,\ldots,x_n) \geq F(x_1,\ldots, 0,\ldots,x_n)$ for all possible input assignments. In other words, switching x_i from 0 to 1 cannot make *F* change from 1 to 0. A Boolean function *F* is said to be negative unate on variable x_i if $F(x_1,\ldots,0,\ldots,x_n) \geq F(x_1,\ldots,1,\ldots,x_n)$ for all possible input assignments. A variable in the support of *F* that is neither positive unate nor negative unate is a binate variable. A function *F* is unate if all of its variables are unate. A function *F* is a positive (negative) unate function if all its variables are positive (negative) unate. If there is at least one binate variable in the support of *F*, then *F* is a binate function.

In this work, $\cdot$, +, $\overline{bar}$ and $\rightarrow$ denote logical conjunction (AND), disjunction (OR), complementation (NOT) and material implication (IMP) operations. In several cases the '$\cdot$' operator is just omitted for better text format so that $(x_1 \cdot x_2) = (x_1x_2)$.

A literal is a variable (positive literal) or a complemented variable (negative literal). The conjunction of literals is a cube. A cube comprising only positive literals is a positive cubes, whereas a cube comprising only negative literals is a negative cube and a cube comprising both positive and negative literals is a binate cube. A minterm is a cube comprising all input variables of F. A cube c is an implicant cube (or simply implicant) of F if $c = 1$ implies $F \neq 0$. An implicant c is an implicant prime (or simply a prime) of F if removing any literal from c produces a cube that is not an implicant. Equivalently, a prime is a maximal implicant. A prime is an essential prime if there is at least one minterm that is only covered by this prime.

2.2 BOOLEAN FUNCTION REPRESENTATION AND OPTIMIZATIONS

2.2.1Truth Table and Karnaugh Map

A truth table is the most straightforward way to represent a Boolean function where the values for all possible input assignments $\mathbf{x}$ are displayed. A Karnaugh map, in turn, is similar to a truth table with the difference that the data is displayed over a matrix. The main problem of both representations is related to scalability. Both truth tables and Karnaugh map always require 2^n positions, regardless of the function itself. A truth table can be represented as a hexadecimal integer where the most significant bit of such an integer is defined by minterm $x_0 x_1 \dots x_{n-1}$.

2.2.2 Sum-of-products (and two-level minimization)

A disjunction of implicant cubes of F corresponds to a sum-of-products (SOP) f for F such that $F_{ON} \subseteq f_{ON} \subseteq (F_{ON} \cup F_{DC})$. The set of cubes in f represents a cover ζ for F. If removing any element from ζ leads to $F_{ON} \nsubseteq f_{ON}$ then ζ is an irredundant cover. If all cubes in ζ are prime, then ζ is a prime cover. A prime and irredundant cover is an irredundant sum-of-products (ISOP). In other words, if any cube or literal is removed from f, then s f is not a cover for F, then f is an ISOP. The cardinality of a SOP is the number of cubes in it. A minimum SOP presents the smallest cardinality among all covers for F. A minimal SOP is not a proper superset of any other SOP. Notice that a SOP always represents a CSF. If the target function F is an ISF, a SOP covering F represents a CSF F_2 such that $F_2(\mathbf{x}) = 1$ implies $F(\mathbf{x}) \neq 0$ and $F_2(\mathbf{x}) = 0$ implies $F(\mathbf{x}) \neq 1$. A function F can also be written in a product-of-sums (POS) form. Similarly to an ISOP, a POS is irredundant (IPOS) if any sum and any literal can be removed from the expression without modifying the target function.

A usual question is to determine which of the SOP or POS expressions is the most compact for representing a given function. Equivalently, the question could be which of F and $\bar{F}$ have the smallest SOP representation. Notice that, if there is a POS with k sums representing a given function F, then there is a SOP representation for $\bar{F}$ containing k cubes, through De Morgan's theorem. The decision to represent F or $\overline{F}$ is important because, if the optimal SOP for F has *k* cubes with *m* literals each, then the optimal SOP for $\overline{F}$ can have up to m^kcubes (SASAO, 2001). However, given a SOP f , there is not a simple way to decide whether the SOP $\bar{f}$ comprises more or less cubes than *f*.

Even though the number of cubes in the list grows theoretically as 2^n, in practice this number is much smaller. For instance, an *n*-input AND function requires only one cube with *n* literals. In this sense, a SOP representation for a 50-input AND is perfectly reasonable, whereas a truth table (or Karnaugh map) representation is unfeasible.

Two-level minimization is the process of optimizing a SOP expression. The goal is to obtain a SOP comprising the smallest number of cubes (*i.e.*, a minimum SOP). There are both exact and heuristic minimization procedures. In this work, we are mostly interested in the heuristic approaches because these ones present a good trade-off between the solution quality and the execution time. A minimum solution comprises the smallest number of cubes among all possible solutions. Notice that a minimal solution is not a proper superset of any other solution. In other words, a minimal solution cannot be improved by simply removing a literal or a cube from it.

Typical tasks in heuristic two level minimization algorithms are the expansion and reduction of cubes. The goal of a cube expansion is to transform implicants into primes. Moreover, as an implicant is expanded it may cover other implicants that may be discarded. During the expansion process it is necessary to check if the expanded cube still represents an implicant. The cube reduction transforms primes into implicants. As a cube is reduced, this cube can become dominated by another cube and so be removed from the SOP. During a cube reduction, it is necessary to check if the resulting SOP still covers the target function. Typically, two-level minimizers usually perform a loop based on expansion and reduction processes. The loop execution stops when the solution cannot be no more improved.

To illustrate the process of two-level minimization, consider a function *F* described by the Karnaugh map shown in Fig. 2.1(a). Assume an initial cover for the function corresponding to the highlighted cubes being written as follows:

$$f^* = \overline{x_0}\,\overline{x_2} + \overline{x_0}\,x_1 + x_0x_2 + x_0\,\overline{x_1} \qquad (2.2.1)$$

Notice that (2.2.1) comprises only primes because no cube from it can be expanded without modifying the described function. Moreover, no prime can be removed from (2.2.1) without modifying the function. One can obtain another solution by reducing cube $\overline{x_0}x_1$ to $\overline{x_0}x_1x_2$, as illustrate in Fig. 2.1(b). Furthermore, it can be verified that replacing $\overline{x_0}\,x_1$ by $\overline{x_0}x_1x_2$ leads to a valid solution because $\overline{x_0}x_1\overline{x_2}$ is dominated by $\overline{x_0}\,\overline{x_2}$. The new cover can be written as follows:

$$f^* = \overline{x_0}\,\overline{x_2} + \overline{x_0}\,x_1x_2 + x_0x_2 + x_0\,\overline{x_1} \tag{2.2.2}$$

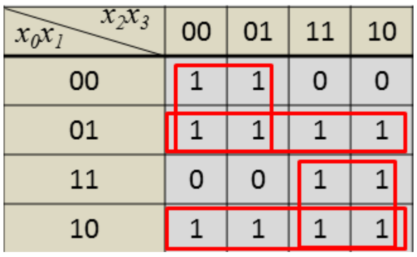
Even though (2.2.2) comprises one more literal than (2.2.1), the cardinality of both coverings is the same. The next step is to expand cube $\overline{x_0}x_1\,x_2$ to cube $x_1\,x_2$, as illustrate in Fig. 2.1(c), giving the following expression:

$$f^* = \overline{x_0}\,\overline{x_2} + \;x_1x_2 + x_0x_2 + x_0\,\overline{x_1} \tag{2.2.3}$$

In (2.2.3), the cube x_0x_2 is redundant and so can be removed. After this removal, the covering comprises one less cube than the original solution (2.2.1). The final covering is illustrated in Fig. 2.1(d), and is written as follows:

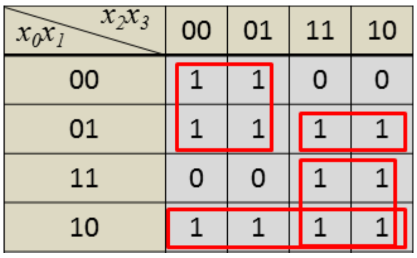
$$f^* = \overline{x_0}\,\overline{x_2} + \;x_1x_2 + x_0\,\overline{x_1} \tag{2.2.4}$$

Figure 2.1 – Different coverings $f^* = \overline{x_0}\,\overline{x_2} + \;x_1x_2 + x_0\,\overline{x_1}$: (a) initial, (b) after reduction of cube $\overline{x_0}x_1$ to $\overline{x_0}x_1\,x_2$, (c) after expanding $\overline{x_0}x_1x_2$ to x_1x_2, and (d) final minimum covering.



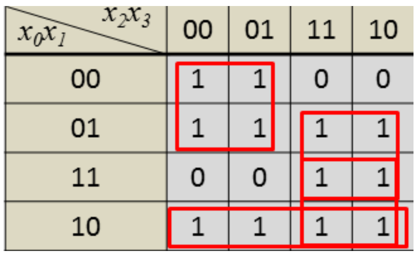
$x_0x_1 \backslash x_2x_3$	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	0	0	1	1
10	1	1	1	1

(a)



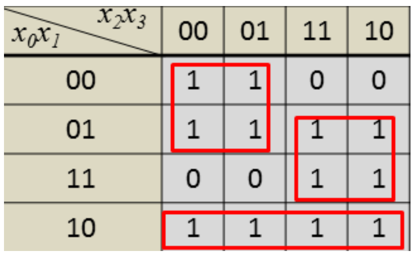
$x_0x_1 \backslash x_2x_3$	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	0	0	1	1
10	1	1	1	1

(b)



$x_0x_1 \backslash x_2x_3$	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	0	0	1	1
10	1	1	1	1

(c)



$x_0x_1 \backslash x_2x_3$	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	0	0	1	1
10	1	1	1	1

(d)

Source: The author.

2.2.2.1 SOP of Unate Functions

Unate functions are very interesting for two level minimizations because if *F* is a unate function, then all primes are essential and the ISOP is unique. In other words, the ISOP comprises all primes of *F* and only the primes of *F*. Therefore, given a unate covering for a unate function *F*, the ISOP for *F* can be obtained by simply removing cubes that are not prime. This process is known as single cube containment. For instance, consider a function defined by the following expression:

$$f^* = x_1\, x_2 + \overline{x_3}\, x_4 + x_1 x_2 \overline{x_3} \tag{2.2.5}$$

In (2.2.5), x_1 and x_2 are positive unate variables whereas x_3 and x_4 are negative unate variables. It follows that *F* is a unate function. To verify if (2.2.5) is an ISOP, we check for cubes that are redundant. In this case, cube $x_1 x_2 \overline{x_3}$ is redundant because it is covered by $x_1\, x_2$. Therefore, $x_1 x_2 \overline{x_3}$ can be removed, leading to the following unique ISOP:

$$f^* = x_1\, x_2 + \overline{x_3}\, x_4 \tag{2.2.6}$$

2.2.3 Binary Decision Diagram

Boolean functions can also be represented through binary decision diagram (BDD) (LEE,1959), (AKERS, 1978). A BDD is a rooted, directed, acyclic graph. A node of a BDD can be either a decision node or a terminal node. Each decision node represents a Boolean variable and each terminal node is either 1 or 0. Each non-terminal node has a high and a low child. A BDD node is redundant if the low and high children are the same or if there is another node that represents the same function. Moreover, a BDD is reduced (RBDD) if there are no redundant nodes. Furthermore, a BDD is ordered (OBDD) if, for all pairs of variables x_i and x_j, x_i and x_j appear in the same order for any path comprising both x_i and x_j. A reduced and ordered BDD (ROBDD) is a canonical representation of the corresponding Boolean function (BRYANT, 1986).

2.2.4 Factored forms, factoring and division

A factored form can be recursively defined as a literal (*i.e.*, x_i or $\overline{x_i}$), as well as a conjunction of factored forms or a disjunction of factored forms (BRAYTON, 1982). In other words, a factored form is a Boolean expression where only literals can be complemented.

Hence, $x_1(\overline{x_2}+x_3)$ is a factored form whereas $x_1\overline{(\overline{x_2}+x_3)}$ is not. Similarly, $(\overline{x_1}+\overline{x_2})$ is a factored form whereas $\overline{x_1 x_2}$ is not, both representing the 2-input NAND.

Factoring is the process of obtaining a factored form representing a Boolean function. The main goal of factoring algorithms is to reduce the number of literals in the resulting factored form. Factoring algorithms can be classified as algebraic or Boolean. An algebraic factoring algorithm considers Boolean variables as integers and applies standard algebra. The main limitation of algebraic factoring is that the notion of complement does not exist. Therefore, literals x_0 and $\overline{x_0}$ are treated as independent variables. Boolean factoring uses properties that are specific to Boolean algebra. Such properties include the following relations: $x_0\overline{x_0}=0$, $x_0+\overline{x_0}=1$ and $x_0+x_0x_1=x_0$.

Boolean methods tend to yield better results but are usually more complex than algebraic methods. To illustrate the differences between these methods, we take a function *F* written as follows:

$$f^* = x_1x_2 + x_2x_3 + \overline{x_1}x_3 \qquad (2.2.7)$$

An algebraic factoring method can obtain one of the following expressions:

$$f^* = x_2(x_1 + x_3) + \overline{x_1}x_3 \qquad (2.2.8)$$

$$f^* = x_1x_2 + x_3(x_2 + \overline{x_1}) \qquad (2.2.9)$$

Both (2.2.8) and (2.2.9) comprise five literals. A Boolean factoring method, on the other hand, may return the following expression:

$$f^* = (x_1 + x_3)(x_2 + \overline{x_1}) \qquad (2.2.10)$$

which comprises four literals. Notice that, in order to obtain (2.2.10), the property $x_0\overline{x_0}=0$ is applied. Therefore, an algebraic method is not able to obtain (2.2.10) as solution.

The factoring process can also be understood as a division operation over a given SOP f. If f is divided by a SOP d, then f is written as follows:

$$f = qd + r \qquad (2.2.11)$$

where q, d and r are SOP. q is the quotient, d is the divider and r is the remainder. If d is a cube, then d is a single cube divisor. Otherwise, d is a multiple cubes divisor. If d is a cube and r is empty, then d is a factor of f. If f has no factors, then f is a cube-free expression. If d is a cube and q is a cube-free expression, then d is a co-kernel of f and q is a kernel of f (BRAYTON, 1982).

Obtaining single cube divisors can be done using a matrix structure where each column is a literal, and each row is a cube in the input SOP. If cube c_i comprises the literal x_j, a ‘1’ is placed on the corresponding matrix position. A rectangle in the matrix is a set of columns and rows such that all matrix entries corresponding to the intersections of these columns and rows contain a ‘1’. Neither the columns nor the rows have to be continuous. The weight of a rectangle is the improvement obtained (usually measured in terms of reduction on the number of literals) by selecting that rectangle to perform the division operation. A prime rectangle is a rectangle for which the weight can be increased by adding a row or a column. It has been shown that the good divisors for an expression can be identified from the prime rectangles (RUDELL, 1989). For instance, consider an expression f, as follows:

$$f = x_0x_1x_2x_3 + x_0x_1x_4 + x_3x_6x_7 + x_3x_6x_8 \qquad (2.2.12)$$

The resulting matrix is shown in Table 2.1. There are two prime rectangles in the Table 2.1. The first is given by the intersections of columns $\{x_0, x_1\}$ and lines $\{x_0x_1x_2x_3, x_0x_1x_4\}$. The second is given by the intersection of $\{x_3, x_6\}$ and lines $\{x_3x_6x_7, x_3x_6x_8\}$. Hence, the single cube divisors for (2.2.12) are x_0x_1 and x_3x_6. The resulting expression is as follows:

$$f = x_0x_1(x_2x_3 + x_4) + x_3x_6(x_7 + x_8) \qquad (2.2.13)$$

Table 2.1 – Matrix for single cube divisor extraction for (2.2.12).

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
$x_0x_1x_2x_3$	1	1	1	1					
$x_0x_1x_4$	1	1			1	1			
$x_3x_6x_7$				1			1	1	
$x_3x_6x_8$				1			1		1

Source: The author.

Identifying good divisors using this matrix structure resembles a covering algorithm over a Karnaugh map in the sense that prime rectangles are analogous to prime implicants. However, the prime rectangle size does not need to be a power of 2 and the matrix positions do not need to be adjacent.

The process for identifying good multiple cubes divisors uses the same matrix structure. However, the columns and rows have different meanings. Each column is a cube in a kernel of f while each row is a co-kernel of f. The candidates co-kernels to be adopted in the division process can be obtained from the intersection of the pairs of cubes in *f*. This process is illustrated in the following:

$$f = x_0x_1x_2x_3 + x_0x_1x_4 + x_0x_1x_5 + x_2x_3x_7 + x_4x_7 \tag{2.2.14}$$

The first pair of cubes $\{x_0x_1x_2x_3, x_0x_1x_4\}$ leads to the co-kernel x_0x_1 and the kernel $x_2x_3 + x_4$. Hence, there is a line x_0x_1 and two columns x_2x_3 and x_4 in the matrix. On the other hand, there is no intersection of literals regarding cubes $x_0x_1x_2x_3$ and x_4x_7. Therefore, this pair does not generate any candidate divisors. The resulting matrix is shown in Table 2.2. A prime rectangle consists of the intersection of columns $\{x_2x_3, x_4\}$ and rows $\{x_0x_1, x_7\}$. When f is divided by $x_2x_3 + x_4$, we obtain the following expression:

$$f = (x_2x_3 + x_4)(x_0x_1 + x_7) + x_0x_1x_5 \tag{2.2.15}$$

Table 2.2 – Matrix for multiple cubes divisor extraction for (2.2.14).

	x_2x_3	x_4	x_5	x_0x_1	x_7
x_0x_1	1	1	1		
x_2x_3				1	1
x_4				1	1
x_7	1	1			

Source: The author.

Notice that, even though there are two prime rectangles presented in Table 2.2, only one of them is selected because both rectangles lead to the same cubes. The rectangles are transpositions of each other. However, if rectangles overlay, there are cases where it can be useful to allow redundancies. In order to illustrate possible benefits of redundancy, consider the matrix shown in Table 2.3.

Table 2.3 – Matrix to exemplify the need for redundancy.

	x_3	x_4	x_5	x_6
x_0	1	1	1	1
x_1	1	1	1	
x_2	1			1

Source: The author.

The first prime rectangle is given by the intersection of lines $\{x_0, x_1\}$ and columns $\{x_3, x_4, x_5\}$. The second prime rectangle is given by the intersection of lines $\{x_0, x_2\}$ and columns $\{x_3, x_6\}$. Hence, the rectangles overlap. If redundancy is not allowed, then only one of the rectangles can be selected. Let the first rectangle be selected, then the resulting expression contains 10 literals, as follows:

$$f = (x_0 + x_1)(x_3 + x_4 + x_5) + x_6(x_0 + x_2) + x_2x_6 \tag{2.2.16}$$

On the other hand, if redundancy is allowed, we obtain an expression with nine literals, as follows:

$$f = (x_0 + x_1)(x_3 + x_4 + x_5) + (x_3 + x_6)(x_0 + x_2) \tag{2.2.17}$$

Even though cube x_0x_3 appears in both terms of (2.2.17), the resulting factored form comprises one less literal than one represented by (2.2.16).

One limitation of such matrix based method is that the number of rows and columns in the matrix are quadratic functions on the number of cubes. Different works have proposed methods to reduce the set of candidate divisors. One criteria applied is to set a maximum bound on the number of literals that a divisor can have (MODI, 2004).

2.3 FUNCTIONAL COMPOSITION

Functional composition (FC) is a bottom-up approach to logic synthesis (MARTINS, 2012). The main idea of FC is to obtain expressions for complex functions by combining known solutions for simpler functions. For this reason, a set of basic functions for which optimal solutions are known must be defined. FC was proposed to perform factoring of Boolean functions and has also been applied to different emerging logic paradigms that are not necessarily based on AND and OR operations (MARTINS, 2014), (MARTINS, 2015), (NEUTZLING, 2014).

When FC is used to perform factoring, the basic functions correspond to the literals and complemented ones. These basic functions are combined through AND and OR operators, yielding functions for which the optimal factored form comprises two literals. Since expressions with m literals are created before expressions with $m + 1$ literals, the first expression found for a function is also the optimal solution. In order to check whether an expression in the first solution for a function, a functional representation, such as a truth table, is stored together with each created expression. To illustrate the idea of FC, we consider the synthesis of a three input function given by the following SOP:

$$f = x_0x_1 + x_0x_2 \tag{2.3.1}$$

The truth table for (2.3.1) is shown in Table 2.4. The equivalent integer is E0.

Table 2.4 – Truth table for $f = x_0x_1 + x_0x_2$.

x_0	x_1	x_2	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Source: The author.

Since F is a three input function, the set of basic functions is as shown in Table 2.5. None of the expressions in Table 2.5 is a solution to (2.3.1). Therefore, we associate all pairs of expressions in Table 2.5 through AND and OR operators. The resulting expressions are shown in Table 2.6 together with the respective integer representation of the truth table.

Since none of the expressions in Table 2.6 represents the target function, expressions with three literals have to be generated. For this, each expression in Table 2.5 is combined with an expression in Table 2.6 through AND and OR operations. When expressions x_0 and $(x_1 + x_2)$ are combined through an AND operation, we obtain the following expression:

$$f = x_0(x_1 + x_2) \qquad (2.3.2)$$

The truth table of (2.3.2) is E0. Therefore, (2.3.2) is an expression for the target function. Since (2.3.2) is the first solution found, it is also an optimal solution.

The FC process can be improved by considering properties of Boolean functions. In particular, let F_1 and F_2 be two functions such that $F_1 \subseteq F$ and $F_2 \subseteq F$, then $(F_1F_2) \subseteq F$. Therefore, performing an AND operation between two functions that are dominated by F does not help the FC to approach to a solution. A similar argument is valid for OR operation. In the previous example, function x_0 dominates the target function. Hence, only AND operations using x_0 are useful.

Table 2.5 – Basic functions for FC using three input variables.

x_0	x_1	x_2	$f1$	$f2$	$f2$	$f3$	$f4$	$f5$
0	0	0	0	1	0	1	0	1
0	0	1	0	1	0	1	1	0
0	1	0	0	1	1	0	0	1
0	1	1	0	1	1	0	1	0
1	0	0	1	0	0	1	0	1
1	0	1	1	0	0	1	1	0
1	1	0	1	0	1	0	0	1
1	1	1	1	0	1	0	1	0
Integer			F0	0F	CC	33	AA	55
Boolean expression			x_0	$\overline{x_0}$	x_1	$\overline{x_1}$	x_2	$\overline{x_2}$

Source: The author.

Table 2.6 – Expressions with two literals and the respective truth table represented as integer.

Expression	Truth table		Expression	Truth table
$x_0 + x_1$	FC		$x_0 x_2$	A0
$x_0 + \overline{x_1}$	F3		$x_0 \overline{x_2}$	50
$\overline{x_0} + x_1$	CF		$\overline{x_0} x_2$	0A
$\overline{x_0} + \overline{x_1}$	F3		$\overline{x_0}\,\overline{x_2}$	05
$x_0 x_1$	C0		$x_1 + x_2$	EE
$x_0 \overline{x_1}$	30		$x_1 + \overline{x_2}$	DD
$\overline{x_0} x_1$	0C		$\overline{x_1} + x_2$	BB
$\overline{x_0}\,\overline{x_1}$	03		$\overline{x_1} + \overline{x_2}$	77
$x_0 + x_2$	FA		$x_1 x_2$	88
$x_0 + \overline{x_2}$	F5		$x_1 \overline{x_2}$	44
$\overline{x_0} + x_2$	AF		$\overline{x_1} x_2$	22
$\overline{x_0} + \overline{x_2}$	5F		$\overline{x_1}\,\overline{x_2}$	11

Source: The author.

2.4 PROBABILITY OF BOOLEAN FUNCTIONS

The probability of a CSF F ($p(F)$) is the probability that a random input assignment x satisfies $F(x) = 1$. By definition, $P(1) = 1$ and $P(0) = 0$. If F is represented as a BDD with x_i as root node, then $p(F)$ can be defined as:

$$p(F) = p(x_i)P\left(F_{high}\right) + (1 - p(x_i))P(F_{low}) \tag{2.4.1}$$

To other representations, computing the probability is not always as straightforward. Since the probability of a function does not depend on the representation used, we can consider the probability of a SOP. For instance, consider a SOP f with $|f|$ cubes. Let f_j denote the sum of the first j cubes in f such that $f_{|f|} = f$. The probability of a f_i is given by:

$$p(f_i) = P(f_{i-1}) + P(c_i) - P(c_i f_{i-1}) \tag{2.4.2}$$

To evaluate (2.4.2), term $P(c_i f_{i-1})$ must be computed. Notice that $c_i f_{i-1}$ is a SOP that can comprises up to $|f| - 1$ cubes. The evaluation of $P(c_i F_{i-1})$ is also done using (2.4.2) and may lead to the need of evaluating the probability $P(c_i c_j F_{i-1})$ of a SOP with $|f| - 2$ cubes. The process continues until a SOP with a single cube is reached. Overall, the total number of cubes visited to evaluate $p(f)$ grows exponentially with $|f|$. In this sense, the most promising approach is to transform the SOP into a BDD. Even though the resulting number of nodes in a BDD can be an exponential function on the number of cubes of the SOP, BDD tends to be a more compact representation than SOP.

3 RESISTIVE SWITCHING DEVICES

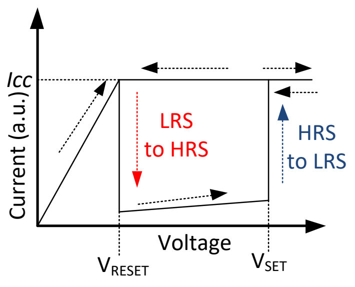
In this Chapter, we discuss the electrical behavior of RSD as well as applications of RSD to perform logic.

3.1 ELECTRICAL BEHAVIOR OF RESISTANCE SWITCHING DEVICES

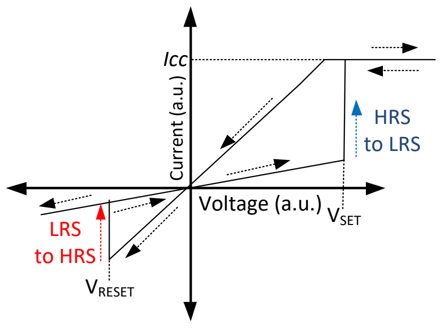
The most common type of RSD presents two resistive states. The device can be either in a low resistance state (LRS), with a resistance R_{ON}, or in the high resistance state (HRS), with a resistance R_{OFF}.

RSD can be classified as bipolar or unipolar device (PAN, 2014). In a unipolar device the resistance state depends only on the magnitude of the applied voltage. In this case, the effect related to the resistance state is expected to be dominated by some thermal effect like Joule heating (WOUTERS, 2015). In contrast, in a bipolar device, the transitions from the HRS to LRS and on the opposite sense occur with voltage biasing on different polarities. Therefore, the RS mechanism should be field driven (WOUTERS, 2015). The idealized I-V curves for quasi-static operation are shown in Fig. 3.1(a) and in Fig. 3.1(b) for unipolar and bipolar RSD, respectively. A unipolar device, initially in the LRS, switches to the HRS when the voltage bias reaches V_{RESET}. Then, the device remains in this state until the voltage bias reaches V_{SET}, when the device switches back to a LRS. Finally, if the voltage biasing is removed, the device stays in the LRS. A compliance current (Icc) is taken into account in order to avoid a destructive transition from HRS to LRS. A bipolar device, initially in the HRS, switches to LRS when a voltage bias of V_{SET} is reached. Transition from the LRS to HRS is only observable when a voltage V_{RESET} with opposite polarity is applied. Even though, a positive biasing causes the transition from HRS to LRS, as depicted in Fig. 3.1(b), several devices require a negative bias to transition from HRS to LRS.

Figure 3.1 – Ideal I-V curves for different types of RSD: (a) unipolar and (b) bipolar.



(a)



(b)

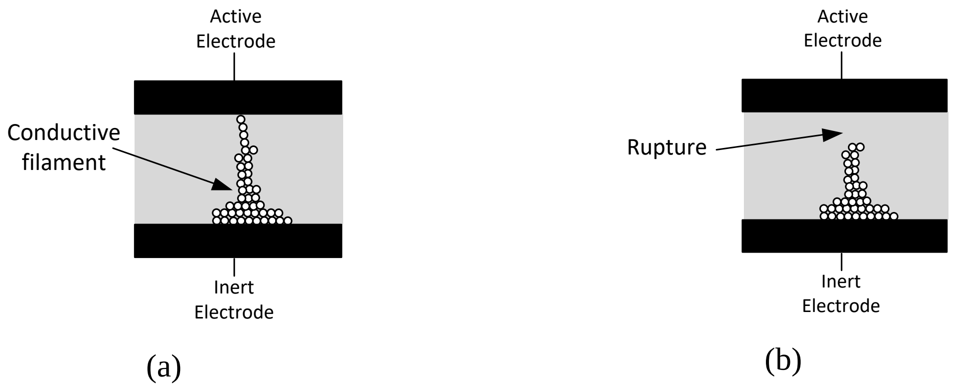
Source: (WOUTERS, 2015).

The resistance switching effect can occur due to several physical mechanisms, which can be used to classify the devices. The main variations of RSD are electrochemical metallization cell (ECM), valence change (VC), phase change (WOUTERS, 2015), (PAN, 2014) and magnetic devices (KENT, 2015).

The basic structure of ECM cells consists of one electrode made from an active metal (e.g. Ni, Ag or Cu), one electrode made from an inert metal (e.g. W, Au or Ir) and an ion-conducting insulating solid electrolyte (WOUTERS, 2015), (PAN, 2014).

During the transition from the HRS to the LRS, a high voltage biasing is applied to the device. The resulting electrical field makes cations from the active electrode move into the insulator. The metal cations drift through the insulator and are reduced near to the inert electrode by electrons injected through the inert electrode. This process continues, creating a conductive filament (CF) and connecting both electrodes, as illustrated in Fig. 3.2(a). The subsequent resistive switching behavior is a consequence of dissolution (RESET process), illustrated in Fig. 3.2(b), or formation (SET process) of the CF. In most cases, only one CF is formed. Hence, the ON resistance becomes nearly independent from the device area since the CF size is very small compared to the total area. This is an interesting property for scaling since the ratio R_{OFF}/R_{ON} should increase as the device shrinks.

Figure 3.2 – Electrochemical memristive device: (a) RSD in LRS with a CF, and (b) RSD in HRS with ruptured CF.

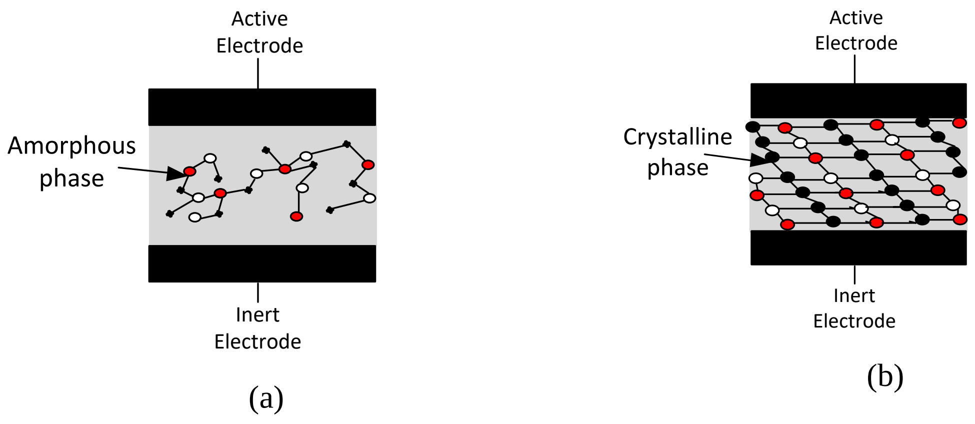


Source: (PAN, 2014).

In valence change devices, the main cause of resistive switching is the transport of oxygen vacancies (Vo) (WOUTERS, 2015), (PAN, 2014). The main structure for this kind of devices consists on an insulator layer made of a transition metal oxide (*e.g.*, TiO_2-x), where x is used to denote the existence of Vo, and two electrodes. Under a voltage biasing, Vo drifts through the insulator and accumulates near to the negatively biased electrode. Eventually, a bridge of Vo contacting both electrodes is formed. The high concentration of Vo causes the transition to a conducting state.

Resistive switching on phase change devices is based on the transition of a phase change material from an amorphous phase, which presents a high resistance value, to a crystalline phase with low resistance (WOUTERS, 2015), (PAN, 2014). The amorphous and crystalline phases are shown, respectively, in Fig. 3.3(a) and in Fig. 3.3(b). The RESET process is obtained by applying a high electric current for a short time, so quickly removing the current. The electric current melts the crystalline phase and, when the electric current is quenched, the material transitions to the amorphous phase. For the SET process, a smaller current is applied during a longer time. The current anneals the phase change material to a temperature between the crystallization and the melting temperatures. Since Joule heating plays a major role on the resistive switching, phase change devices are unipolar.

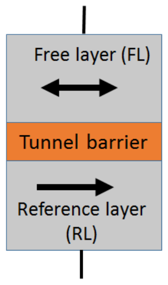
Figure 3.3 – Phase change memristive device: (a) highly resistive amorphous state, and (b) crystalline state with small resistance.



Source: (PAN, 2014).

The basic element in magnetic RSD is the spin-transfer-torque magnetic tunnel junction (STT-MTJ). A STT-MTJ comprises two ferromagnetic layers and an insulator layer (tunnel barrier), as depicted in Fig. 3.4. One of the ferromagnetic layers is a free layer (FL), meaning that its magnetization can be modified. The other ferromagnetic layer is a reference layer (RL), meaning that its magnetization is constant. The FL can be either in a parallel (PP) or antiparallel (AP) state with respect to the RL. In the PP (AP) state, the MTJ resistance is small (large). This effect is known as tunnel magnetoresistance (TMR).

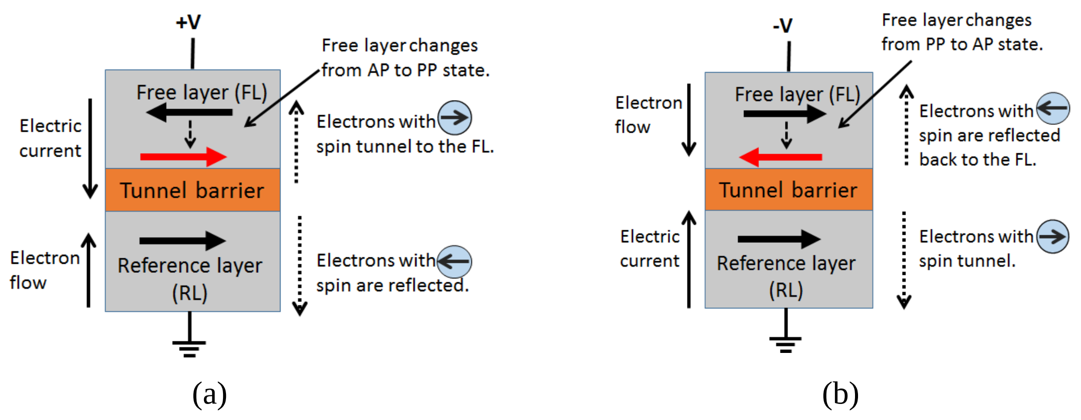
Figure 3.4 – Basic structure of STT-MTJ.



Source: (KENT, 2015).

The state of a given STT-MTJ can be modified through the spin-transfer-torque (STT) effect spin injection. By applying a positive voltage to the FL (+V) electrons flow from the RL to the FL, electrons with opposite spin with respect to the reference layer are reflected while electrons with spin aligned to the RL tunnel to the FL. As shown in Fig. 3.5(a), tunneled electrons cause a spin accumulation in the FL so causing its magnetization to switch to PP state,. When a negative biasing is applied to the FL (-V), electrons flow from the FL to the RL. Electrons with spin aligned to the RL pass through the device while electrons with opposite spin are reflected back to the FL. Thus, the FL magnetization switches to AP state, as depicted in Fig. 3.5(b).

Figure 3.5 – Spin-transfer-torque effect in a magnetic tunnel junction: (a) from anti-parallel to parallel, and (b) from parallel to anti-parallel.


+V
Free layer changes from AP to PP state.
Electric current
Free layer (FL)
Electrons with spin tunnel to the FL.
Tunnel barrier
Electron flow
Reference layer (RL)
Electrons with spin are reflected.
(a)
-V
Free layer changes from PP to AP state.
Electron flow
Free layer (FL)
Electrons with spin are reflected back to the FL.
Tunnel barrier
Electric current
Reference layer (RL)
Electrons with spin tunnel.
(b)

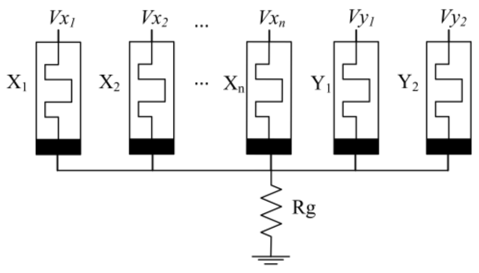
Source: The author.

3.2 MATERIAL IMPLICATION LOGIC

3.2.1 Overview

The RSD-IMP logic structure was firstly described in (BORGHETTI, 2010) and is one of the most studied approaches to perform logic-in-memory using RSD. The basic circuit topology is illustrated in Fig. 3.6, where several RSD are connected to a load resistor Rg, with resistance value Rg. The devices X_1 to X_n are input RSDs whereas the devices Y_1 and Y_2 are work RSDs. Notice that there may be more than two work RSDs. An input RSD stores an input variable, and the work RSDs are used to save intermediate computation values. For each RSD X_k (Y_k), its control voltage is denoted by V_{xk} (V_{yk}) and its state, which represents a Boolean variable, is denoted by x_k (y_k). We adopt the convention that R_{ON} represents the logic value 1 whereas R_{OFF} represents the logic value 0. In the following, we consider the RSD as bipolar device with very high R_{OFF}/R_{ON} ratio. Notice that part of this discussion can be extrapolated to other devices (SUN, 2011), (MAHMOUDI, 2013).

Figure 3.6 – Basic topology for RSD-IMP logic structure.



Source: (BORGHETTI, 2010).

The RSD state represents a binary value and logic computation is performed by setting the control voltages to appropriate levels. In particular, the possible values are V_{ON}, V_{COND} and V_{OFF}. V_{ON} is a voltage larger than the positive threshold V_{SET}, V_{OFF} is a voltage smaller than the negative voltage V_{RESET} and V_{COND} is a positive voltage smaller than V_{SET} that has negligible impact on the resistance value. Considering a stand-alone RSD connected to Rg, V_{ON} drives the device into the LRS and V_{OFF} resets the device to the HRS.

The correct behavior of the circuit depends on choosing adequate values for parameters Rg, V_{OFF}, V_{ON} and V_{COND} (KVATINSKY, 2014), (ZHU, 2013). A typical value for Rg is an intermediate value between R_{ON} and R_{OFF}, as the following:

$$Rg = \sqrt{R_{ON}R_{OFF}} \tag{3.2.1}$$

From the previous definitions of V_{COND}, V_{ON} and V_{OFF}, we can derive a set of basic constraints for V_{COND}, V_{ON} and V_{OFF}. For sake of simplicity, we refer to a device RSD X$_i$ whereas the same analysis holds for a RSD Y$_i$. The basic constraints are the following:

1) If V_{COND} is applied to a single RSD X$_i$, with the remaining RSD left at high impedance, then the resulting voltage biasing across X$_i$ is smaller than V_{SET}. Since V_{COND} is a positive voltage defined to be smaller than V_{SET}, this constraint is already respected. Any positive voltage bias smaller than V_{SET} has no impact on the state of the RSD.
2) If V_{ON} is applied to a single RSD X$_i$ at the HRS, the resulting voltage biasing across X$_i$ must be larger than V_{SET}, as given by the following equation:

$$V_{ON}\left(1-\frac{Rg}{Rg+R_{OFF}}\right) \geq V_{SET} \tag{3.2.2}$$

3) If V_{OFF} is applied to a single RSD X$_i$ at the LRS, the resulting voltage biasing across X$_i$ must be smaller than V_{RESET}, as given by the following equation:

$$V_{OFF}\left(1-\frac{Rg}{Rg+R_{OFF}}\right) < V_{RESET} \tag{3.2.3}$$

3.2.2 Logic and electrical behavior

The existence of a common node means that the bias voltage applied to a RSD also depends on the state of other RSDs as well as on the voltages applied to these devices (BORGHETTI, 2010). The basic behavior of this topology is described in the following. If X$_1$ is in the HRS ($x_1 = 0$), then V_{x1} has negligible influence on the circuit, and V_{ON} is able to set Y$_1$ to the LRS (y_1=1). On the other hand, if X$_1$ is in the LRS ($x_1 = 1$), then V_{x_1} increases the voltage across resistor Rg, so resulting in a voltage drop in Y$_1$ insufficient to cause a change of state. Hence, y_1 remains with the previous value. Table 3.1 shows the final values of y_1 ('*next* y_1' column) as function of the initial values x_1 and y_1. The logic behavior shown in Table 3.1 corresponds to the material implication (IMPLY) function ($x_1 \rightarrow y_1 = \overline{x_1} + y_1$). In RSD-IMP logic structure, this operation is known as single input implication because there is only one term at the left hand side. Every time an operation $x_1 \rightarrow y_1$is performed, the resulting value is stored in Y$_1$, overwriting the initial value y_1.

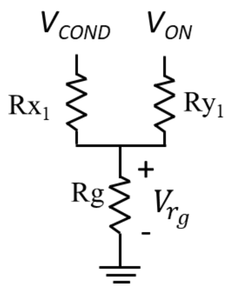
Table 3.1 – Truth table for material implication function ($x_1 \rightarrow y_1 = \overline{x_1} + y_1$).

x₁	*y₁*	*next y₁*
0	0	1
0	1	1
1	0	0
1	1	1

Source: The author.

The expected logic behavior imposes more constraints on the possible values of V_{COND}, V_{OFF} and V_{ON}, in addition to those previously described. Consider that V_{COND} is applied to X_1 while V_{ON} is applied to Y_1 , with the remaining devices at the high impedance state. The equivalent circuit is shown in Fig. 3.7, where Rx_1 and Ry_1 are the equivalent resistance of RSD X_1 and Y_1, respectively. The resistance values of Rx_1 and Ry_1 are given, respectively, by Rx_1 and Ry_1. The voltage across resistor Rg is denoted by V_{r_g}.

Figure 3.7 – Equivalent resistive circuit for IMP operation.



Source: The author.

To understand how the two RSD interact, we use superposition to write V_{r_g} as a function of voltage V_{COND}. An expression for the equivalent resistive voltage divider is the following:

$$V_{r_g} = V_{COND} \frac{Rx_1 + Rg//Ry_1}{Rg//Ry_1} \tag{3.2.4}$$

where $Rg//Ry_1$ is the equivalent resistance of the parallel association of resistors Rg and Ry_1. Each of Rx_1 and Ry_1 can be either R_{ON} or R_{OFF}, leading to four possible cases.

However, since voltage V_{ON} can only cause a transition from HRS to LRS, we only need to evaluate the case when $Ry_1 = R_{OFF}$. Hence, (3.2.4) becomes the following:

$$V_{r_g} = V_{COND}\frac{Rg//R_{OFF}}{Rx_1 + Rg//R_{OFF}} \tag{3.2.5}$$

If $R_{x_1} = R_{OFF}$, then Y_1 should switch to the LRS. Since $V_{COND} < V_{ON}$, this constraint is already included in (3.2.2). Conversely, if $R_{x_1} = R_{ON}$, then Y_1 should remain in the HRS. In this case, the following relationship must be valid:

$$V_{ON} - V_{r_g} < V_{SET} \tag{3.2.6}$$

By combining (3.2.5) and (3.2.6), and replacing R_{x_1} by R_{ON}, we obtain the following constraint:

$$V_{ON} - V_{COND}\left(\frac{Rg//R_{OFF}}{R_{ON} + Rg//R_{OFF}}\right) < V_{SET} \tag{3.2.7}$$

The IMP operation can be expanded to include several terms at the left-hand side such that it is possible to compute the following operation in a single cycle:

$$(x_{i1} + \cdots + x_{ik}) \to y_1 \tag{3.2.8}$$

An operation in the form of (3.2.8) is a multi-input implication operation (SHIN, 2011). A multi-input implication is performed by applying V_{COND} to all $X_{i1}, \ldots,$ Xik while V_{ON} is applied to Y_1. If at least one of $X_{i1}, \ldots, X_{ik}$, is in the LRS, then the voltage across Rg increases, as given by (3.2.7) such that Y_1 does not switch from the HRS to the LRS. Notice that as the number of terms in (3.2.8) increases, the equivalent resistance seen by each RSD decreases and the influence of V_{COND} on V_{r_g} is reduced. Therefore, if we want to use *k* RSD in a multi-input implication, (3.2.7) must be rewritten as:

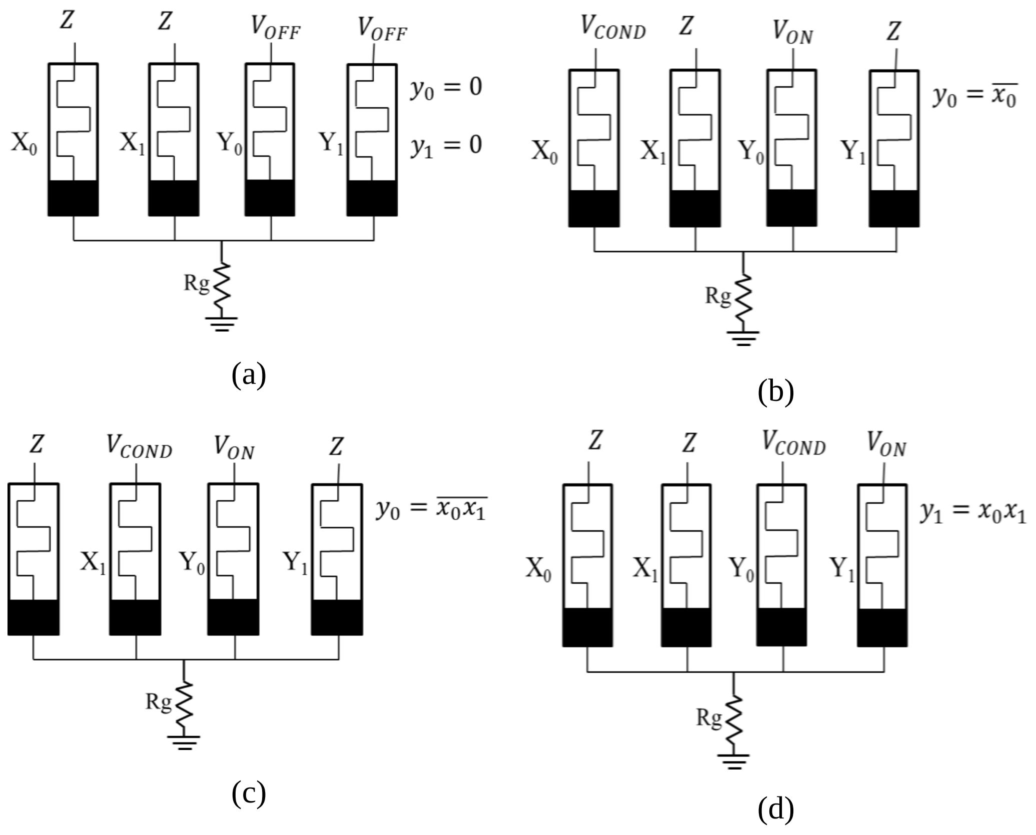
$$V_{ON} - V_{COND}\left(\frac{Rg//R_{OFF_k}}{R_{ON} + Rg//R_{OFF_k}}\right) < V_{SET} \tag{3.2.9}$$

where R_{OFF_k} is the equivalent resistance of k parallel RSD in the HRS and $Rg//R_{OFF_k}$ is the equivalent resistance of the parallel association of R_g and R_{OFF_k}. Even though (3.2.9) depends on the number of RSDs used in a multi-input implication, this is not a major concern for RSD with a sufficiently large R_{OFF}/R_{ON} resistance ratio because $R_{OFF_k}//R_g \approx R_g$.

In the following, we illustrate the behavior of RSD-IMP logic structure using the sequence of instructions for the AND2 function shown in Table 1.2.1.

Example 3.2.1: In Fig. 3.8 it is summarized the cycles to perform the AND operation, where Z denotes that a device is left in the high impedance state. The first cycle resets both Y_0 and Y_1 to the HRS by applying V_{OFF} to both Y_0 and Y_1, while the remaining RSD are left in the high impedance state, as illustrated in Fig. 3.8(a). In the second cycle, shown in Fig. 3.8(b), V_{COND} is applied to X_0 while V_{ON} is applied to Y_0. If $x_0 = 0$, then the impact of V_{COND} on the V_{Rg} is small. Therefore, V_{ON} causes Y_0 to switch to the LRS. Otherwise, V_{COND} makes V_{Rg} to increase, such that $V_{ON} - V_{Rg} < V_{SET}$. In this case, Y_0 remains in the HRS. During the third cycle, V_{COND} is applied to X_1 while V_{ON} is applied to Y_0, as depicted in Fig. 3.8(b). The circuit behavior is similar to the one observed during the second cycle. Notice that if Y_0 was driven to the LRS during the second cycle, the third cycle does not change the state of Y_0. After the third cycle, Y_1 is in the LRS if at least one of x_0 and x_1 is 0. Hence, the value in y_0 is given by $y_0 = \overline{x_0} + \overline{x_1} = \overline{x_0 x_1}$. Finally, in the last cycle, V_{COND} is applied to Y_0 while V_{ON} is applied to Y_1. Y_1 changes to the LRS only if Y_0 is in the HRS, as illustrated in Fig. 3.8(d). After this cycle, the values of y_1 and y_0 are the complement of each other. Therefore, the final value of y_1 is given by the AND of x_0 and x_1. When desired, the state of Y_1, which represents the final value of x_0x_1, can be read by applying a voltage biasing to Y_1 and measuring the voltage or the current across Rg.

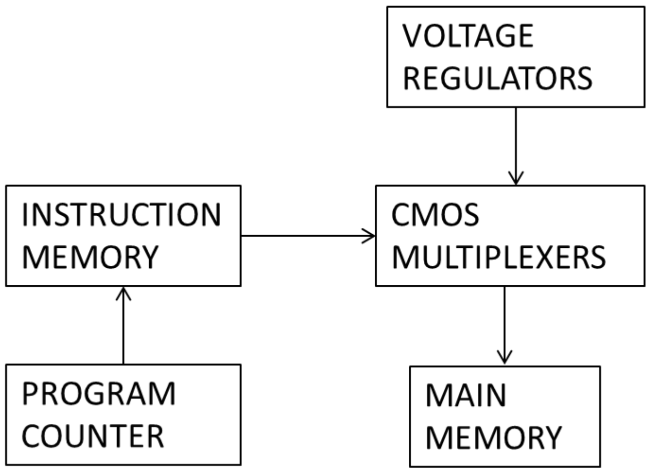
Figure 3.8 – Voltages for each instruction to evaluate the AND2 function: (a) reset cycle; (b) instructions to compute $y_0 = \overline{x_0}$; (c) instructions to compute $y_0 = \overline{x_0 x_1}$; (d) last instructions to compute $y_1 = x_0 x_1$.



Source: The author.

Similarly, to all RSD-based approaches for logic-in-memory structure, RSD-IMP requires a control block which is illustrated in Fig. 3.9 (RAHMAN, 2016). Voltage regulators generate the voltages V_{OFF}, V_{ON} and V_{COND} . A instruction memory stores the sequence of instructions to evaluate a target function. The instructions serve as control for CMOS transmission gate based multiplexers such that the different control voltages V_{COND}, V_{ON} and V_{OFF} are applied as data inputs of the multiplexers. Hence, the instructions select which voltage is applied to each device or if the device should be left in the high impedance state. The main memory block is the RSD-NVM where the logic computation is performed and the program counter is used to access the following instructions. Given that the use of a RSD-NVM as standard memory (*i.e.*, without logic capability) requires voltage regulators and multiplexers, it appears that the overhead caused by adding the memory capability is the extra RSD-NVM required to store the instructions and the program counter.

Figure 3.9 – Schematic of the architecture for RSD-IMP logic structure.



Source: The author.

3.2.3 Logic synthesis for RSD-IMP logic structure

In RSD-IMP logic structure, the computation of a given Boolean function is performed as a sequence of multi-input implication and reset operations. In this sense, the main task of the logic synthesis for RSD-IMP logic structure is to find a sequence of operations to evaluate the behavior of the function from an initial representation form. At the same time, there is an optimization challenging related to find the smallest sequence of operations that can be computed using a maximum of $n + m$ devices.

To aid the synthesis process, it is important to define classes of expressions which have a direct relationship to a sequence of instructions. In particular, we aim to define a set of expressions $\mathcal{F}$ such that, given an expression $f \in \mathcal{F}$ for some function *F*, *f* can be directly translated to a sequence of instructions *S* for *F*. The size of *S* should also be directly related to the size of *f*. In this sense, optimizing the size of *f*, also optimizes *S*. Finally, if *S* requires $n + m$ RSD to be evaluated, then we say that *f* can be evaluated with $n + m$ RSD. An example of such class is the recursive Boolean form (RBF), discussed in the following.

3.2.3.1 Recursive Boolean forms

The most studied class of expressions for RSD-IMP logic structure is the class of recursive Boolean forms. A RBF *f* can be defined as follows:

$$f = f_0 + \overline{(f_1 + \overline{(f_2 + \overline{\ldots + \overline{(f_{\phi-2} + \overline{f_{\phi-1}})}})})} \qquad (3.2.10)$$

where each f_i is a SOP and ϕ is the number of levels in f. In RSD-IMP logic structure, each f_i is a unate function. Since we are considering multi-input implications, it is useful to restrict each f_i to be a negative unate SOP. In this manner, each cube in a f_i can be computed as a single multi-input implication operation. Notice that some previous works consider that an input RSD stores the complement of variable (*i.e.*, X$_i$ contains $\overline{x_i}$ instead of x_i) (POIKONEN, 2012), (TEODOROVIC, 2013). In this case, each f_i is a positive unate SOP. In this sense, we find more straightforward to consider that an input X$_i$ RSD stores x_i and we restrict the cubes in f_i to be negative unate. We also write a RBF in the expanded form. Considering an odd value for ϕ, (3.2.10) can be expanded as follows:

$$f = f_0 + \overline{f_1}f_2 + \overline{f_1}\overline{f_3}f_4 + \cdots + \overline{f_1}\overline{f_3} \ldots \overline{f_{\phi-2}}f_{\phi-1} \qquad (3.2.111)$$

The negative unate restriction reduces the number of functions that (3.2.10) can represent. Equation (3.2.10) only represents functions for which the minterm $\overline{x_0}\,\overline{x_1} \ldots \overline{x_{n-1}}$ is not part of the offset. Otherwise, the complement of the function is applied, as follows:

$$\bar{f} = \overline{f_0}f_1 + \overline{f_0}\overline{f_2}f_3 + \overline{f_0}\overline{f_2} \ldots \overline{f_{\phi-3}}\,f_{\phi-2} + \overline{f_0}\overline{f_2} \ldots \overline{f_{\phi-3}\,f_{\phi-1}} \qquad (3.2.12)$$

Example 3.2.2: Consider the 3-input exclusive-NOR (XNOR3) function, written as follows:

$$f = \overline{x_0}\,\overline{x_1}\,\overline{x_2} + \overline{x_0}x_1x_2 + x_0\overline{x_1}x_2 + x_0x_1\overline{x_2} \qquad (3.2.13)$$

An RBF for the XNOR3 is the following:

$$f = f_0 + \overline{f_1}f_2 \qquad (3.2.14a)$$

$$f_0 = \overline{x_0}\,\overline{x_1}\,\overline{x_2} \qquad (3.2.14b)$$

$$f_1 = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} + \overline{x_1}\,\overline{x_2} \qquad (3.2.14c)$$

$$f_2 = \overline{x_0} + \overline{x_1} + \overline{x_2} \qquad (3.2.14d)$$

If the target function is the 2-input exclusive-OR (XOR3), the aforementioned challenges arises because the minterm $\overline{x_0}\,\overline{x_1}\,\overline{x_2}$ is part of the offset. Therefore, the RBF for the XOR3 is the complement of the XNOR3 RBF, as given by the following:

$$f = \overline{f_0}f_1 + \overline{f_0}\,\overline{f_2} \tag{3.2.15a}$$

$$f_0 = \overline{x_0}\,\overline{x_1}\,\overline{x_2} \tag{3.2.15b}$$

$$f_1 = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} + \overline{x_1}\,\overline{x_2} \tag{3.2.15c}$$

$$f_2 = \overline{x_0} + \overline{x_1} + \overline{x_2} \tag{3.2.15d}$$

Notice that all f_i are the same for the XNOR3 and XOR3 cases. In order to differentiate both RBFs, we say that the XNOR3 RBF has a positive phase whereas the XOR3 RBF has a negative phase, meaning that the XOR3 RBF must be complemented to yield the correct result.

A sequence of operations to evaluate F in RSD-IMP logic structure can be directly obtained from a RBF, as described in (TEODOROVIC, 2013). Each cube in the RBF becomes a multi-input implication while reset and complement operations are added between levels. For instance, a sequence of operations derived from (3.2.14) for the XNOR3 function is shown in Table 3.2. Steps 2 to 4 evaluate f_2, steps 6 to 8 are obtained from f_1, and step 11 is used for f_0. Notice that the evaluation is performed from f_2 to f_0. In Section 4.1, we propose a scheme to evaluate the RBF from f_0 to f_2 and show the benefits of using this order.

Table 3.2 - Sequence of operations to evaluate the XNOR3 function.

1.	RESET(y_1, y_2)	$y_1 = 0, y_2 = 0$
2.	$x_0 \rightarrow y_2$	$y_2 = \overline{x_0}$
3.	$x_1 \rightarrow y_2$	$y_2 = \overline{x_0} + \overline{x_1}$
4.	$x_2 \rightarrow y_2$	$y_2 = \overline{x_0} + \overline{x_1} + \overline{x_2}$
5.	$y_2 \rightarrow y_1$	$y_1 = x_0x_1x_2$
6.	$(x_0 + x_1) \rightarrow y_1$	$y_1 = x_0x_1x_2 + \overline{x_0}\,\overline{x_1}$
7.	$(x_0 + x_2) \rightarrow y_1$	$y_1 = x_0x_1x_2 + \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2}$
8.	$(x_1 + x_2) \rightarrow y_1$	$y_1 = x_0x_1x_2 + \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} + \overline{x_1}\,\overline{x_2}$
9.	RESET(y_2)	$y_2 = 0$
10.	$y_1 \rightarrow y_2$	$y_2 = \overline{x_0}x_1x_2 + x_0\,\overline{x_1}x_2 + x_0x_1\,\overline{x_2}$
11.	$(x_0 + x_1 + x_2) \rightarrow y_2$	$y_2 = \overline{x_0}\,\overline{x_1}\,\overline{x_2} + \overline{x_0}x_1x_2 + x_0\,\overline{x_1}x_2 + x_0x_1\,\overline{x_2}$

Source: (TEODOROVIC, 2013).

For sake of simplicity, in Table 3.2 it is shown the equivalence between the sequence of instructions and the target function through the obtained expression. However, we have verified all sequence of operations shown herein using an ROBDD-based approach. A multiple-output ROBDD is used to store the state of all RSDs used, where the BDD outputs are the states of the RSDs. After all operations are performed, the ROBDD of the output RSD is compared to the ROBDD of the target function. Hereafter, this approach is used regardless of the method used to obtain the sequence of instructions.

3.2.4.2 Logic synthesis methods for RSD-IMP logic

Different logic synthesis methods have been proposed to minimize RBF. In the following, we discuss some of these methods. Two cover based methods are presented in (POIKONEN, 2012) and in (RAGHUVANSHI, 2014). Both algorithms try to find a covering using primes containing only negative literals. Since a traditional covering is not always possible with this restriction, both methods switch between covering F_{ON} and F_{OFF} until a constant function is obtained. Since the methods are similar, we consider a single example for both.

Example 3.2.3: In the following, we consider the synthesis of the XOR2 function based on the Karnaugh map representation, as shown in Fig. 3.10(a).

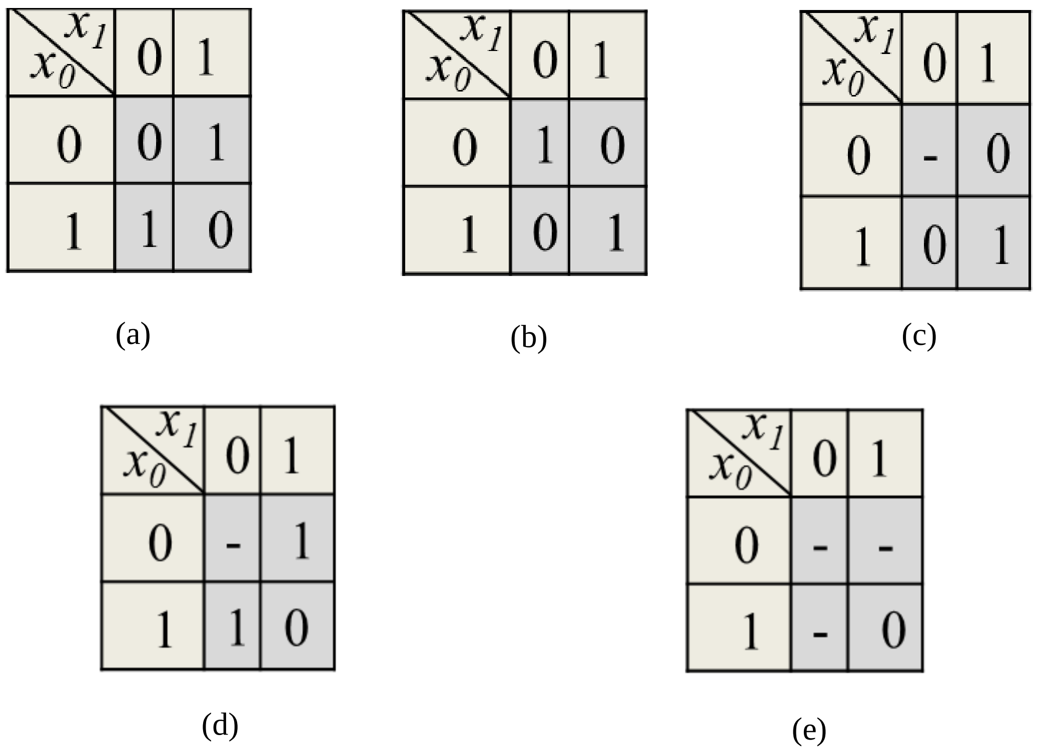
The only primes for F_{ON} are $\overline{x_0}x_1$ and $x_0\overline{x_1}$. Since neither of these cubes are negative, the algorithms search for a covering for $\bar{F}$, represented in Fig. 3.10(b). The two primes for $\bar{F}$ are x_0x_1 and $\overline{x_0}\,\overline{x_1}$. Therefore, the cube $\overline{x_0}\,\overline{x_1}$ is added to f_0 and the minterms covered by $\overline{x_0}\,\overline{x_1}$ are set to *don't care* value, as illustrated in Fig. 3.10(c). Notice that this process results in a new function $F1$. Since there are no more negative cubes that can cover $F1$, the algorithm searches for a covering for $\overline{F1}$, illustrated in Fig. 3.10(d). Both cubes $\overline{x_0}$ and $\overline{x_1}$ are added to f_1 and the covered minterms are set to *don't care*, as shown in Fig. 3.10(e). The execution of the algorithm terminates since the resulting function is a constant. The final RBF is the following:

$$f = \bar{f}_0 f_1 \tag{3.2.16a}$$

$$f_0 = \overline{x_0}\,\overline{x_1} \tag{3.2.16b}$$

$$f_1 = \overline{x_0} + \overline{x_1} \tag{3.2.16c}$$

Figure 3.10: Synthesis process for the XOR2 function: (a) original function representation, (b) complemented function, (c) resulting function *F1* after selecting the cube x_0x_1, (d) function $\overline{F1}$, and (e) final map after selecting the cubes x_0 and x_1.



$x_0 \backslash x_1$	0	1
0	0	1
1	1	0

(a)

$x_0 \backslash x_1$	0	1
0	1	0
1	0	1

(b)

$x_0 \backslash x_1$	0	1
0	-	0
1	0	1

(c)

$x_0 \backslash x_1$	0	1
0	-	1
1	1	0

(d)

$x_0 \backslash x_1$	0	1
0	-	-
1	-	0

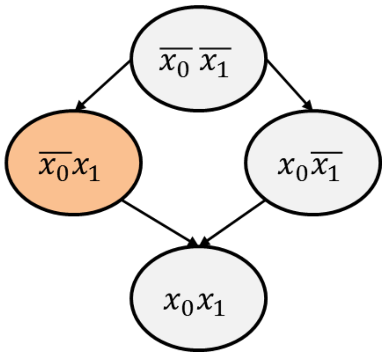
(e)

Source: The author.

The algorithms presented in (POIKONEN, 2012) and in (RAGHUVANSHI, 2014) differ on the manner that they decide to go from F_{ON} to F_{OFF} (and *vice-versa*). The algorithm presented in (POIKONEN, 2012) always chooses the largest positive prime to add to the target expression, regardless of whether the prime covers F_{ON} or F_{OFF}. In contrast, the algorithm presented in (RAGHUVANSHI, 2014) only changes between F_{ON} and F_{OFF} when all possible positive primes are selected.

The method proposed in (TEODOROVIC, 2013) begins by creating a directed graph where each vertex corresponds to a minterm. For a minterm c_i, π_i represents the product of negative literals in c_i. A directed edge from c_j to c_k is created if all literals π_k are in π_j and there is exactly one literal that appears in π_j but not in π_k. Then, the graph is colored according to the function value for each minterm. The initial graph for $f = \overline{x_0}\, x_1$ is shown in Fig. 3.11, where white and orange nodes represent, respectively, minterms resulting in 0 and 1.

Figure 3.11 – Graph for function $f = \overline{x_0}\, x_1$.



Source: The author.

The next step is to partition the graph which is done by a greedy algorithm. Each partition corresponds to a level of the RBF. The root of the graph is set to partition p_0 and is set as the partition root. The algorithm then visits the other nodes of the graph using breadth-first search (BFS). A node c_i is placed in p_k if it has the same color of the partition root and if there is no path to c_i from any other node c_j that is not in any partition. Once all nodes are visited, the algorithm traverses the tree, using BFS, until a node with a different color from the partition root that is not allocated to any partition is found. This node becomes the root of a new partition and the process is repeated. When all vertices have been assigned to a partition, the algorithm execution stops.

The root of the first partition p_0 is vertex $\overline{x_0}\,\overline{x_1}$. Vertex $x_0\overline{x_1}$ is added to the same partition because it is also black, and all arriving edges at $x_0\overline{x_1}$ come from a vertex that is already assigned to a partition. The next node visited is $\overline{x_0}x_1$, which is not added to the current partition because it has a different color from the root node. Finally, the vertex x_0x_1 is visited. Even though this node has the same color as the root, x_0x_1 is not assigned to the current partition because there is a path from $\overline{x_0}x_1$, that is a node that does not belong to any partition, to x_0x_1. The next partition has $\overline{x_0}x_1$ as root. Node x_0x_1 is not added to this new partition because it is from a different color. Finally, a third partition for node x_0x_1 is created. The resulting partitions are as follows:

$$p_0 = \{\overline{x_0}\,\overline{x_1}, x_0\overline{x_1}\} \tag{3.2.17a}$$

$$p_1 = \{\overline{x_0}x_1\} \tag{3.2.17b}$$

$$p_2 = \{x_0x_1\} \tag{3.2.17c}$$

Once the partitions are created, the algorithm removes unnecessary nodes. These removed nodes are the nodes that have a successor within the partition. For the given example, the vertex $\overline{x_0}\,\overline{x_1}$ is removed because $x_0\overline{x_1}$ is a successor in the same partition. Each partition p_i corresponds to a level f_i in the RBF. Moreover, for each c_j in p_i, π_j is a cube in f_i. Finally, if the nodes in p_0 are orange (*i.e.*, represent a 1), the RBF phase is positive. Otherwise, the RBF phase is negative. Therefore, the resulting RBF for $f = \overline{x_0}\,x_1$ is the following:

$$f = \overline{f_0}f_1 \tag{3.2.18a}$$

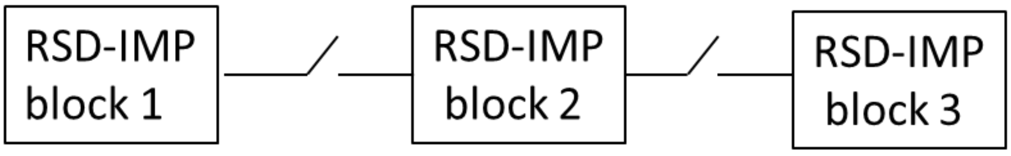
$$f_0 = \overline{x_1} \tag{3.2.18b}$$

$$f_1 = \overline{x_0} \tag{3.2.18c}$$

3.2.5 Alternative structures

One of the main constraints in RSD-IMP logic structure is that, in its original form, a single instruction can be executed per cycle. In this sense, one approach to improve the performance of RSD-IMP logic structure is to modify the standard topology shown in Fig. 3.6, to allow for operations to occur in parallel. Usually, such modification consists of adding transistors between different RSD-IMP blocks (KIM, 2011), (KVATINSKY, 2014). When the transistors are *off*, the RSD-IMP blocks are electrically isolated from each other, and so can perform operations in parallel. When the transistors are *on*, the different blocks can interact with each other. The topology is illustrated in Fig. 3.12, where each RSD-IMP block presents the internal topology as the one shown in Fig. 3.6.

Figure 3.12 – Parallel structure for RSD-IMP logic structure.



Source: (KIM, 2011).

The method described in (KIM, 2011) receives a SOP as input. The variables in the input SOP are copied to the different blocks. Then, all cubes are evaluated in parallel by letting the transistors in the *off* state. Finally, the cubes are summed. For the last step, all transistors are set to the *on* state. If f has $|f|$ cubes and the maximum number of positive literals in a cube is ρ, then the number of cycles to evaluate the SOP (λ_f) and the number of RSDs used (m_f) are, respectively:

$$\lambda_f = 5 + 2n + \rho \tag{3.2.19}$$

and

$$m_f = |f|(n + 2) + 2 \tag{3.2.20}$$

An improvement to the work described in (KIM, 2011) is presented in (XIE, 2017) where it is noticed that, in some cases, the transistor that isolate the blocks are not required. In (XIE, 2017), the minterms of the function are evaluated in parallel and summed afterwards. To obtain the parallelism, each minterm is evaluated in a different line of the RSD crossbar. This approach allows any single-output Boolean function to be evaluated using only seven cycles. However, the number of RSD (m_f) used to evaluate a function with *M* minterms is the following:

$$m_f = (2n + 1)(M + 1) \tag{3.2.21}$$

Another logic synthesis approach targeting parallel architectures is a BDD-based method described in (CHAKRABORTI, 2014). The method configures the different blocks such that all nodes in a BDD level are evaluated in parallel. Hence, if L_m is the maximum number of nodes in any level, then L_m RSD-IMP blocks are used. The number of cycles to evaluate the BDD (λ_f) and the number of RSDs used (m_f) are, respectively, the following:

$$\lambda_f = 6n + C_{edges} \tag{3.2.22}$$

and

$$m_f = 5L_m + F_{max} + C_{edges} \tag{3.2.23}$$

where C_{edges} is the maximum number of complemented edges in any level of the BDD, and F_{max} is the maximum total fanout of any level in the BDD.

Other variations are presented in (ZHU, 2013) and in (ZHANG, 2015). In these works, the goal is to increase the set of basic operations by adding one control voltage. In particular, a control voltage $V_{COND_{NEG}}$ is added such that if $V_{COND_{NEG}}$ is applied to X_0 while V_{OFF} is applied to Y_0, then the next state of Y_0 (y_0') is 1 only if both x_0 and y_0 are 1. Hence, y_0' is given by the AND operation of x_0 and y_0:

$$y_0' = x_0 y_0 \tag{3.2.24}$$

However, both references (ZHU, 2013) and (ZHANG, 2015) do not present an algorithm to exploit the AND operation.

It may be worth to notice that several works present logic computation techniques and topologies using RSDs but without describing a clear logic synthesis procedure to attain the expected solution (AMIRSOLEIMANI, 2017), (ALAMGIR, 2016), (LEEVY, 2014), (TALATI, 2016). As a consequence, in the context of this work, this fact makes difficult the comparison between different approaches presented in the literature..

4 SYNTHESIS AND EVALUATION OF RECURSIVE BOOLEAN FORMS

As discussed in Chapter 3, RBF is a usual form to represent Boolean functions when targeting RSD-IMP logic structure because RBF can always be translated into a sequence of operations computable with two work RSDs. In this chapter, we propose two novel approaches to synthesize RBF and present improvements regarding the evaluation of such forms in RSD-IMP logic structure.

4.1 EVALUATION OF RECURSIVE BOOLEAN FORMS

We discuss two contributions regarding the evaluation of RBF in RSD-IMP logic. Initially, we discuss a more efficient method to obtain a sequence of operations from a given RBF. This novel sequence of operations allows us to perform short circuit evaluation (SCE) over RBF.

4.1.1 Reverse Evaluation of Recursive Boolean Forms

The conventional evaluation of an RBF f with ϕ levels begins from level $f_{\phi-1}$, as discussed in Section 3.2.4. In this work, we propose an evaluation method that begins from level f_0. In order to derive the proposed scheme, we write a positive phase RBF f as follows:

$$f = f_0 + \overline{f_1}f_2 + \overline{f_1} \cdot \overline{f_3}f_4 + \cdots + \overline{f_1} \cdot \overline{f_3} \ldots \cdot \overline{f_{\phi-2}}f_{\phi-1} = \tau_0 + \tau_1 + \cdots + \tau_{\phi-1/2} \tag{4.1.1}$$

where ϕ is an odd integer and every term τ_i is in the form:

$$\tau_i = fp_i \cdot fn_i = f_{2i} \cdot \prod_{j=0}^{\left\lceil \frac{i}{2} \right\rceil} \overline{f_{2j+1}} = f_{2i} \cdot \overline{f_1} \cdot \overline{f_3} \cdot \ldots \overline{f_{2i-1}} \tag{4.1.2}$$

For each term τ_i, fp_i is the positive part of the term and fn_i is the negative part of τ_i. Notice that, for all τ_i, fn_i comprises all f_k such that k is an odd integer, and $k < j$. Moreover, fn_i consists of fn_{i-1} with the inclusion of $\overline{f_{2\iota-1}}$. For instance, consider the following RBF f:

$$f = f_0 + \overline{f_1}f_2 + \overline{f_1} \cdot \overline{f_3}f_4 \tag{4.1.3a}$$

$$f_0 = \overline{x_0} \tag{4.1.3b}$$

$$f_1 = \overline{x_1} + \overline{x_2} \tag{4.1.3c}$$

$$f_2 = \overline{x_3} \tag{4.1.3d}$$

$$f_3 = \overline{x_4} + \overline{x_5} \tag{4.1.3e}$$

$$f_4 = \overline{x_6} \tag{4.1.3f}$$

There are three τ_i in (4.1.3), namely τ_1, τ_2 and τ_3. These terms are given as follows:

$$\tau_0 = f_0 = \overline{x_0} \tag{4.1.4a}$$

$$\tau_1 = \overline{f_1} f_2 = \overline{(\overline{x_1} + \overline{x_2})}\,\overline{x_3} \tag{4.1.4b}$$

$$\tau_2 = \overline{(\overline{x_1} + \overline{x_2})}\ \overline{(\overline{x_4} + \overline{x_5})}\,\overline{x_6} \tag{4.1.4c}$$

If we take the term τ_2 in (4.1.4c), the respective fn_2 and fp_2 are given as follows:

$$fp_2 = \overline{x_6} \tag{4.1.5a}$$

$$fn_2 = \overline{(\overline{x_1} + \overline{x_2})}\ \overline{(\overline{x_4} + \overline{x_5})} \tag{4.1.5b}$$

When evaluating a term τ_i, the first step is to store the complement of fn_i ($\overline{fn_i}$) into Y1. The term $\overline{fn_i}$ is given by:

$$\overline{fn_i} = f_1 + f_3 + \cdots + f_{2i-1} \tag{4.1.6}$$

Since fn_i comprises fn_{i-1} when evaluating τ_i, the state of Y1, given by y_1, can be written as follows:

$$y_1 = f_1 + f_3 + \cdots + f_{2i-1} \tag{4.1.7}$$

Therefore, only f_{2i-1} must be added to Y1. The second step is to compute each cube in τ_i. Let f_{2i} be written in SOP form, as follows:

$$f_{2i} = c_1 + c_2 + \cdots + c_m \tag{4.1.8}$$

where each c_j is a negative cube, given by:

$$c_j = \overline{x_{j1}}\,\overline{x_{j2}} \ldots \overline{x_{j\gamma}} \tag{4.1.9}$$

A multi-input implication can be used to evaluate each $(fn_i c_j)$, as follows:

$$\left(c_{11} + c_{12} + \cdots + c_{1\gamma} + y_1\right) \rightarrow y_0 \tag{4.1.10}$$

where y_1 is the complement of fn_i, as given by (4.1.7).

Example 4.1.1: The proposed sequence of instructions to compute the XNOR3 function, given by (3.2.14), is shown in Table 4.1. In line 2, the term τ_0 is evaluated. Then, lines 3 to 5 store the complement of fn_1 into y_2, as given by (4.1.6). Finally, lines 6 to 8 are used to evaluate the term $fn_1 fp_1$.

Table 4.1 - Proposed sequence of instructions to evaluate the XNOR3 function.

	Operation	y_1	y_2
1.	y_1=0, y_2=0	0	0
2.	$(x_0 + x_1 + x_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}\,\overline{x_2}$	
3.	$(x_0 + x_1) \rightarrow y_2$		$\overline{x_0}\,\overline{x_1}$
4.	$(x_0 + x_2) \rightarrow y_2$		$\overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2}$
5.	$(x_1 + x_2) \rightarrow y_2$		$\overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} + \overline{x_1}\,\overline{x_2}$
6.	$(x_0 + y_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}\,\overline{x_2} + \overline{x_0}\,x_1\,x_2$	
7.	$(x_1 + y_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}\,\overline{x_2} + \overline{x_0}\,x_1\,x_2 + x_0\,\overline{x_1}\,x_2$	
8.	$(x_2 + y_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}\,\overline{x_2} + \overline{x_0}\,x_1\,x_2 + x_0\,\overline{x_1}\,x_2 + x_0\,x_1\,\overline{x_2}$	

Source: The author.

When comparing the proposed technique with the conventional recursive computation, the number of steps is reduced in three. The reason for this improvement is the elimination of several complement operations. In the conventional method, before the computation of the ith recursion level can begin, a complement operation is performed to the result of the $(i + 1)$th level. Complement operations are costly in IMP RSD logic structure since two cycles are required.

If ϕ is even, then the RBF becomes as follows:

$$f = f_0 + \overline{f_1} f_2 + \overline{f_1} \cdot \overline{f_3} f_4 + \cdots + \overline{f_1} \cdot \overline{f_3} \ldots \overline{f_{\phi-1}} = \tau_0 + \tau_1 + \cdots + \tau_{\phi/2} \qquad (4.1.11)$$

where the last term $\tau_{\phi/2}$ comprises only negative literals, as follows:

$$\tau_{\phi/2} = \overline{f_1} \cdot \overline{f_3} \ldots \overline{f_{\phi-1}} \qquad (4.1.12)$$

In order to evaluate (4.1.11), we consider that the positive part of the last term $\tau_{\phi/2}$ equals 1 (*i.e.*,$fp_{\tau_{\phi/2}} = 1$). In this case, (4.1.10) becomes:

$$y_1 \rightarrow y_0 \qquad (4.1.13)$$

Example 4.1.2: Consider an RBF *f*, as follows:

$$f = f_0 + \overline{f_1} \quad (4.1.14a)$$

$$f_0 = \overline{x_0} \quad (4.1.14b)$$

$$f_1 = \overline{x_1} + \overline{x_2} \quad (4.1.14c)$$

The sequence of instructions for *f* is shown in Table 4.2. After the reset operation, the value of f_0 is stored into y_1. Then, in lines 3 and 4, the complement of fn_1, which is equal to f_1, is stored into y_2. Since the number of levels in the RBF is even, we consider that $f_2 = 1$ and perform the last operation, as given by (4.1.13).

Table 4.2 – Proposed sequence of instructions to evaluate equation (4.1.14).

	Operation	y_1	y_2
1.	y_1=0, y_2=0	0	0
2.	$x_0 \rightarrow y_1$	$\overline{x_0}$	
3.	$x_1 \rightarrow y_2$		$\overline{x_1}$
4.	$x_2 \rightarrow y_2$		$\overline{x_2}$
5.	$y_2 \rightarrow y_1$	$\overline{x_0} + x_1x_2$	

Source: The author.

A negative phase RBF can be written as one of the following forms, depending on whether the number of levels is even or odd:

$$f = \begin{cases} \overline{f_0}f_1 + \overline{f_0}\,\overline{f_2}f_3 + \cdots + \overline{f_0}\,\overline{f_2} \ldots f_{\phi-1}, & \text{if } \phi \text{ is even} \\ \overline{f_0}f_1 + \overline{f_0}\,\overline{f_2}f_3 + \cdots + \overline{f_0}\,\overline{f_2} \ldots \overline{f_{\phi-1}}, & \text{otherwise} \end{cases} \quad (4.1.15)$$

The evaluation of a negative phase RBF is similar to a positive phase RBF. However, the negative part of each term in (4.1.15) consists of all f_{2i}.

Example 4.1.3: The sequence of instructions to evaluate the XOR3 function is given in Table 4.3. The RBF for the XOR3 is given in (3.2.15). Notice that the even levels f_0 and f_2 are stored into auxiliary variable y_2. Moreover, the last operation is given by (4.1.13) since we are evaluating a negative phase RBF with odd number of levels.

Table 4.3 – Proposed sequence of instructions to evaluate the XOR3 function.

	Operation	y_1	y_2
1.	y_1=0, y_2=0	0	0
2.	$(x_0 + x_1 + x_2) \rightarrow y_2$		$\overline{x_0}\,\overline{x_1}\,\overline{x_2}$
3.	$(x_0 + x_1 + y_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}\,x_2$	
4.	$(x_0 + x_2 + y_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}x_2 + \overline{x_0}x_1\,\overline{x_2}$	
5.	$(x_1 + x_2 + y_2) \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}x_2 + \overline{x_0}x_1\,\overline{x_2} + x_0\overline{x_1}\,\overline{x_2}$	
6.	$x_0 \rightarrow y_2$		$\overline{x_0}$
7.	$x_1 \rightarrow y_2$		$\overline{x_0} + \overline{x_1}$
8.	$x_2 \rightarrow y_2$		$\overline{x_0} + \overline{x_1} + \overline{x_2}$
9.	$y_2 \rightarrow y_1$	$\overline{x_0}\,\overline{x_1}x_2 + \overline{x_0}x_1\,\overline{x_2} + x_0\overline{x_1}\,\overline{x_2} + x_0x_1x_2$	

Source: The author.

The proposed algorithm to obtain the sequence of operations from a RBF is shown in Algorithm 4.1.1. We use variable p to store whether the RBF is positive or negative. In line 5 the negative terms are stored into an RSD. In line 7, the positive terms are evaluated. Line 10 is used to handle the case of positive phase RBF with even number of levels, or negative phase RBF with odd number of levels. After the evaluation, y_0 contains the final result.

Algorithm 4.1.1 – Proposed algorithm to obtain a sequence of operations for a RBF.

```
1.  $y_0 = 0, y_2 = 1$
2.  if RBF phase is positive then $p = 1$ else $p = 0$
3.  for $f_i$ from $f_0$ to $f_{\phi-1}$
4.    if ($i$ is even and $p = 1$) or ($i$ is odd and $p = 0$) then
5.      $(x_{i0} + x_{i1} + \cdots + x_{ik}) \rightarrow y_1, \forall\, \overline{x_{i0}}\,\overline{x_{i1}} \dots \overline{x_{ik}} \in f_i$
6.    else
7.      $(x_{i0} + x_{i1} + \cdots + x_{ik} + y_1) \rightarrow y_0, \forall\, \overline{x_{i0}}\,\overline{x_{i1}} \dots \overline{x_{ik}} \in f_i$
8.  end for
9.  if ($p = 0$ and $\phi - 1$ is even) or ($p = 1$ and $\phi - 1$ is odd) then
10.   $y_1 \rightarrow y_0$
11. end if
```

Even though the improvement in terms of cycles is interesting, we do not expect larger gains for more complex functions. Overall, the total number of cubes in an RBF grows faster than the number of complement operations. In this sense, the main advantage of the proposed evaluation scheme is that we can generalize the RBF class to SRBF using the novel scheme, while respecting the lower bound of $n + 2$ RSD, as detailed in Chapter 5. Moreover, the proposed evaluation can be used to exploit SCE, as detailed in the following.

4.1.2 Short Circuit Evaluation of Recursive Boolean Forms

In this section, we demonstrate that our novel evaluation scheme of RBF can greatly benefit from SCE to reduce the average number of cycles to evaluate a given Boolean function. In contrast, the standard evaluation scheme cannot exploit SCE.

From (4.1.1), it can be seen that if $f_0 = 1$ then $f = 1$, regardless the evaluation of the other levels. In turn, if $f_0 = 0$ and $f_1 = 1$, then $f = 0$. Therefore, any level f_i only needs to be evaluated when all $f_k = 0$, where $k < i$. Therefore, the final result of the evaluation is known as soon as any cube evaluates to 1. In opposite, this argument does not hold for the standard evaluation scheme because f_i is evaluated before f_{i-1}.

SCE can be done by reading the state of Y$_0$ or Y$_1$ and deciding whether the computation must proceed. The number of cycles to perform this test dependents on the physical implementation of the circuit, being denoted by λ. Herein, we only allow SCE after the evaluation of a f_i. The average number of cycles to compute *f* starting at *fi* is denoted by k_i, and is given by the following formula:

$$k_i = \begin{cases} |f_i| + \lambda + (1 - p_i)k_{i+1}, & \text{if SCE test at } f_i \quad (4.1.16a) \\ |f_i| + k_{i+1}, & \text{otherwise} \quad (4.1.16b) \end{cases}$$

where p_i is the probability that f_i evaluates to 1. Adding a test at f_i reduces the average number of cycles if the following relationship holds:

$$\lambda < (1 - p_i)k_{i+1} \qquad (4.1.17)$$

Herein, we adopt a greedy approach to determine at which level to perform SCE. At each iteration, we decide the level that leads to the greatest reduction in terms of the number of cycles. The method is shown in Algorithm 4.1.2.

Example 4.1.4: In this example, we evaluate the inclusion of SCE at the sequence of operations for the XNOR3, given in Table 4.1. In the following, we assume $\lambda = 1$. The initial probabilities for f_0 and f_1 are given, respectively, by $p_0=1/8$ and $p_1=1/2$.

Algorithm 4.1.2 – Insertion of SCE.

```
1.  compute $p_i$ and $c_i$ for each $f_i$, $0 \leq i < \phi - 1$
2.  while there is a $f_i$ that satisfies (4.1.17), $0 \leq i < \phi - 1$
3.    for each $f_i$ that satisfies (4.1.17)
4.      $k_i'$=result from (4.1.16a)
5.      $k_i$=result from (4.1.16b)
6.      $\Delta k_i = k_i - k_i'$
7.    end for
8.    add the SCE test at the $f_i$ with largest $\Delta k_i$
9.    update $p_i$ and $k_i$ for each $f_i$
10. end while
```

In turn, the initial average number of cycles to evaluate the XNOR3 starting at f_0, f_1 and f_2 are, respectively, $k_0=7$, $k_1=6$ and $k_2=3$. Notice that we need k_2 to evaluate the benefit of performing SCE at f_1. From (4.1.9), adding SCE f_0 and f_1 changes the average number of cycles to $k_0' = 7.25$ and $k_1' = 5.5$, respectively. Therefore, adding SCE to f_1 is the only option to reduce the average number of cycles.

4.2 SYNTHESIS OF RECURSIVE BOOLEAN FORMS

We propose two methods to synthesize RBF along with an optimization procedure for these recursive forms. The first method, described in Section 4.2.1, is based on functional composition (FC) (MARTINS, 2012). FC is a bottom-up strategy that combines solutions for simple functions in order to obtain a solution for a more complex function.

The second method, described in Section 4.2.2, is a decomposition based approach where a RBF h is written as follows:

$$h = f \circ g \tag{4.2.1}$$

where $\circ$ can be an AND, OR or XOR operator. The proposed method obtains a RBF for h from the RBF for f and g.

In Section 4.2.3, we propose an optimization procedure for RBF. This method aims to reduce the length of the RBF either by removing redundant cubes from the RBF or by moving cubes to different levels. After a cube is moved, it can either become redundant or make another cube redundant. Moreover, if all cubes from h_i are moved to some other level, the levels h_{i-1} and h_{i+1} can be merged.

4.2.1 Functional Composition Based Synthesis

In this section, we apply the concept of FC to synthesize positive phase RBF (FC-RBF). Overall, FC-RBF yields optimal RBF, however, being limited to 4-input functions (MARRANGHELLO, 2015a).

The input of the method is a Boolean description of the target function, such as a truth table. Since the method synthesizes only positive phase RBF, if the minterm $\overline{x_0}\ \overline{x_1} \dots \overline{x_{n-1}}$ is not part of F_{ON}, the method obtains an RBF for $\bar{F}$.

As explained in Section 2.2.5, the FC method requires both a cost function to order the found implementations and a set of basic functions. The cost function is the number of cubes in the RBF. The set of base functions consists of all negative unate cubes. In the following, f^i refers to an RBF with i cubes and $\mathcal{F}^i$ is the set of all f^i RBF. Set $\mathcal{F}^1$ comprises all base functions. The rules to obtain more complex RBF are the following:

$$f^k = \begin{cases} f^1 + f^{k-1} & (4.2.2a) \\ f^1 + \overline{f^{k-1}} & (4.2.2b) \end{cases}$$

where (4.2.2a) adds a cube to a level in the RBF, and (4.2.2b) creates a new level in the RBF.

Example 4.2.1: Consider the synthesis of a 2-to-1 multiplexer, given by the truth table shown in Table 4.4.

Table 4.4 – Truth table of 2-to-1 multiplexer.

x_0	x_1	x_2	F	$\bar{F}$
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

Source: The author.

Since the minterm $\overline{x_0}\ \overline{x_1} \ldots \overline{x_2}$ is not part of F_{ON}, we consider an RBF for $\bar{F}$. In the following, for sake of simplicity, we show only the structural representation for each generated function. Since F is a 3-input function, the base functions are the following:

$$\mathcal{F}^1 = \{\overline{x_0}, \overline{x_1}, \overline{x_2}, \overline{x_0 + x_1}, \overline{x_0 + x_2}, \overline{x_1 + x_2}, \overline{x_0 + \ x_1 + x_2}\} \tag{4.2.3}$$

To generate $\mathcal{F}^2$ we can apply either (4.2.2a) or (4.2.2b) to the elements of $\mathcal{F}^1$. The set of RBF obtained from (4.2.2a) is denoted by $\mathcal{F}^{2a}$ and is given by the following:

$$\begin{aligned}\mathcal{F}^{2a} = \{&\overline{x_0} + \overline{x_1},\ \overline{x_0} + \overline{x_2},\ \overline{x_0} + \overline{x_1}\,\overline{x_2},\ \overline{x_1} + \overline{x_2},\ \overline{x_1} + \overline{x_0}\,\overline{x_2},\ \overline{x_2} + \overline{x_0}\,\overline{x_1},\ \overline{x_0}\,\overline{x_1} \\ &+ \overline{x_0}\,\overline{x_2},\ \overline{x_0}\,\overline{x_1} + \overline{x_1}\,\overline{x_2},\ \overline{x_1}\,\overline{x_2} + \overline{x_0}\,\overline{x_2}\}\end{aligned} \tag{4.2.4}$$

where redundant RBFs, such as $\overline{x_0} + \overline{x_0}\,\overline{x_2}$, have been removed. In turn, by applying (4.2.2b), we obtain the set $\mathcal{F}^{2b}$ that comprises the following RBF:

$$\begin{aligned}\mathcal{F}^{2b} = \{&\overline{x_0} + x_1, \overline{x_0} + x_2,\ \overline{x_0} + x_1 + x_2, \overline{x_1} + x_0,\ \overline{x_1} + x_2, \overline{x_1} + x_0 + x_2, \overline{x_2} + x_0,\ \overline{x_2} \\ &+ x_1, \overline{x_2} + x_0 + x_1, \overline{x_0}\,\overline{x_1} + x_2, \overline{x_0}\,\overline{x_2} + x_1, \overline{x_1}\,\overline{x_2} + x_0\}\end{aligned} \tag{4.2.5}$$

Since no expression in $\mathcal{F}^2 = (\mathcal{F}^{2a} \cup \mathcal{F}^{2b})$ represents $\bar{F}$, the algorithm proceeds to generate $\mathcal{F}^3$. Each expression of $\mathcal{F}^2$ is combined with an expressions of $\mathcal{F}^1$ through (4.2.2a) and (4.2.2b). A solution is obtained by using (4.2b) to combine the following expressions:

$$f^2 = \overline{x_0} + x_2 \tag{4.2.6a}$$

$$f^1 = \overline{x_0}\,\overline{x_1} \tag{4.2.6.b}$$

from which follows that an RBF for $\bar{F}$ is the following:

$$\bar{f} = f_0 + \bar{f_1} f_2 \tag{4.2.7a}$$

$$f_0 = \overline{x_0}\,\overline{x_1} \tag{4.2.7b}$$

$$f_1 = \overline{x_0} \tag{4.2.7c}$$

$$f_2 = \overline{x_2} \tag{4.2.7d}$$

4.2.2 Decomposition Based Synthesis

In this section, we propose a decomposition based approach for the synthesis of RBF (DEC-RBF). Compared to FC-RBF, DEC-RBF presents a compromise between the solution quality and scalability. In this sense, DEC-RBF can be used to synthesize functions with larger number of inputs than FC-RBF. Moreover, DEC-RBF can be applied to several representations of Boolean functions such as SOP, BDD and AIG.

Let h be a Boolean function that can be decomposed into functions f and g, as follows:

$$h = f^{\circ}g \tag{4.2.8}$$

where ° can be and AND, OR or XOR operator. In this section, we develop the rules for each decomposition type. Instead of using primary variables, we write h as a function of the levels of f (f_i) and g (g_i). Moreover, all RBFs have positive phase. A negative phase RBF is written as the complement of a positive phase RBF. The method has time complexity $O(|f||g|)$, where $|f|$ and $|g|$ are the number of cubes in f and g, respectively.

4.2.2.1 AND Decomposition

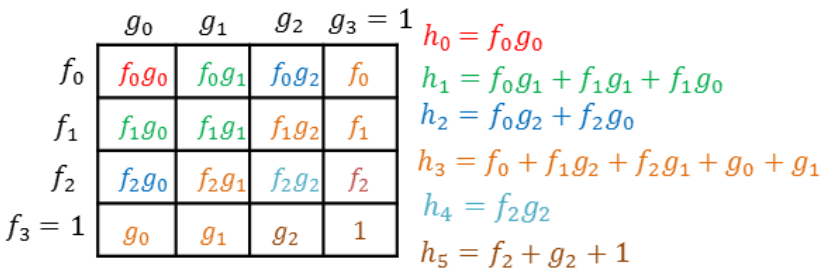
In this section, we consider the case where f and g are an AND decomposition for h (*i.e.*, $h = f \cdot g$). In the following, we denote the number of levels in f and g by φ and γ, respectively. For sake of simplicity, we analyze the case when both φ and γ are odd integers. Moreover, a level is added to both f and g such that $f_{\varphi} = g_{\gamma} = 1$.

The method constructs a matrix, where each f_i and g_j is placed into a row and a column, respectively, and each matrix input is a product $\left(f_i g_j\right)$. Fig. 4.1 illustrates the matrix for the case when $\varphi = \gamma = 3$. The process that selects to each level h_k each product is placed is as follows. Level h_0 comprises only the term f_0g_0. Level h_1 comprises the three neighbors of f_0g_0. Namely, h_1 comprises f_1g_0, f_0g_1 and f_1g_1. To obtain h_2, we move two positions into the matrix from f_0g_0 in the right and down directions. This leads to h_2 comprising f_0g_2 and f_2g_0. Then, the neighbors of f_0g_2 and f_2g_0 are added to h_3. The process continues until all entries are visited.

In the following, we provide a more mathematical view of the algorithm. A Boolean expression h^* that represents h is the following:

$$h^* = f_0 g_0 + f_0 \overline{g_1} g_2 + \cdots + f_0 \overline{g_1} \dots \overline{g_{\gamma-2}}\, g_{\gamma-1} + \cdots + \overline{f_1} f_2 g_0 + \cdots + \overline{f_1} \dots \overline{f_{\varphi-2}}\, f_{\varphi-1} g_0 \tag{4.2.9}$$

Figure 4.1 – Example of matrix for AND decomposition.


g_0 g_1 g_2 $g_3 = 1$
f_0 f_1 f_2 $f_3 = 1$
f_0g_0 f_0g_1 f_0g_2 f_0
f_1g_0 f_1g_1 f_1g_2 f_1
f_2g_0 f_2g_1 f_2g_2 f_2
g_0 g_1 g_2 1
$h_0 = f_0g_0$
$h_1 = f_0g_1 + f_1g_1 + f_1g_0$
$h_2 = f_0g_2 + f_2g_0$
$h_3 = f_0 + f_1g_2 + f_2g_1 + g_0 + g_1$
$h_4 = f_2g_2$
$h_5 = f_2 + g_2 + 1$

Source: The author.

Each term in (4.2.9) comprises two positive literals. From (4.1.1), positive literals should arise from the even levels of h. For this reason, the following relationship holds:

$$h_0 + h_2 + \cdots + h_{\varphi+\gamma} = f_0 g_0 + f_0 g_2 + \cdots + f_0 g_{\gamma-1} + \cdots + f_2 g_0 + \cdots + f_{\varphi-1} g_0 \tag{4.2.10}$$

We define that each level h_{2i} comprises all pairs $f_k g_m$ such that both k and m are even and $k + m = 2i$, we obtain the following expressions:

$$h_0 = f_0 g_0 \tag{4.2.11a}$$
$$h_2 = f_2 g_0 + f_0 g_2 \tag{4.2.11b}$$
$$h_4 = f_4 g_0 + f_2 g_2 + f_0 g_4 \tag{4.2.11c}$$
$$h_{2i} = f_{2i} g_0 + f_{2i-2} g_2 + \cdots + f_2 g_{2i-2} + f_0 g_{2i} \tag{4.2.11d}$$

The next step is to determine the odd levels h_{2i+1}. The odd levels should lead to all negative literals in (4.2.9). An odd level h_{2i+1} can be derived from its antecessor level h_{2i}. For each term $f_p g_r$ in h_{2i}, the terms $f_{p+1} g_r$, $f_p g_{r+1}$ and $f_{p+1} g_{r+1}$ are added to h_{2i+1}. Therefore, the odd levels are as follows:

$$h_1 = f_1 g_0 + f_0 g_1 + f_1 g_1 \tag{4.2.12a}$$
$$h_3 = f_3 g_0 + f_2 g_1 + f_3 g_1 + f_1 g_2 + f_0 g_3 + f_1 g_3 \tag{4.2.12b}$$
$$h_5 = f_5 g_0 + f_4 g_1 + f_5 g_1 + f_3 g_2 + f_2 g_3 + f_3 g_3 + f_1 g_4 + f_0 g_5 + f_1 g_5 \tag{4.2.12c}$$

$$h_{2i+1} = f_{2i+1}g_0 + f_{2i}g_1 + f_{2i+1}g_1 + f_{2i-1}g_2 + f_{2i-2}g_3 + f_{2i-1}g_3 + \cdots + f_3g_{2i-2} + f_2g_{2i-1} + f_3g_{2i-1} + f_1g_{2i} + f_0g_{2i+1} + f_1g_{2i+1} \tag{4.2.12d}$$

The method computes all $f_i \cdot g_j$ where $i \leq \varphi$ and $j \leq \gamma$, and chooses for each product the target level h_k. From (9) and (10), k can be obtained from i and j, as follows:

$$k = \begin{cases} i + j - 1, & \text{if both } i \text{ and } j \text{ are odd} \\ i + j, & \text{otherwise} \end{cases} \tag{4.2.13}$$

Notice that this procedure is also valid to perform a NAND operation between two positive phase RBFs. In such a case, the resulting RBF h has negative phase.

Example 4.2.2: Let h^* be given by:

$$h^* = (\overline{x_0} + x_1)(\overline{x_2} + x_3) \tag{4.2.14}$$

Notice that h^* is an AND decomposition of f^* and g^* which are given by the following equations:

$$f^* = \overline{x_0} + x_1 \tag{4.2.15a}$$

$$g^* = \overline{x_2} + x_3 \tag{4.2.15b}$$

The RBF f and g for f^* and g^* are, respectively:

$$f = f_0 + \overline{f_1} \tag{4.2.16a}$$

$$f_0 = \overline{x_0} \tag{4.2.16b}$$

$$f_1 = \overline{x_1} \tag{4.2.16c}$$

and

$$g = g_0 + \overline{g_1} \tag{4.2.17a}$$

$$g_0 = \overline{x_2} \tag{4.2.17b}$$

$$g_1 = \overline{x_3} \tag{4.2.17c}$$

where φ = γ = 2. All products $f_i g_j$ are given by the following:

$$f_0 g_0 = \overline{x_0}\,\overline{x_2} \tag{4.2.18a}$$

$$f_1 g_0 = \overline{x_1}\,\overline{x_2} \tag{4.2.18b}$$

$$f_2 g_0 = \overline{x_1} \tag{4.2.18c}$$

$$f_0 g_1 = \overline{x_0}\,\overline{x_3} \tag{4.2.18d}$$

$$f_1 g_1 = \overline{x_1}\,\overline{x_3} \tag{4.2.18e}$$

$$f_2 g_1 = \overline{x_3} \tag{4.2.18f}$$

$$f_0 g_2 = \overline{x_0} \tag{4.2.18g}$$

$$f_1 g_2 = \overline{x_1} \tag{4.2.18h}$$

$$f_2 g_2 = 1 \tag{4.2.18i}$$

By applying (4.2.13) to each product $f_i g_j$ in (4.2.18), we obtain the resulting RBF h, as follows:

$$h = h_0 + \overline{h_1} h_2 + \overline{h_1}\,\overline{h_3} \tag{4.2.19a}$$

$$h_0 = f_0 g_0 = \overline{x_0}\,\overline{x_2} \tag{4.2.19b}$$

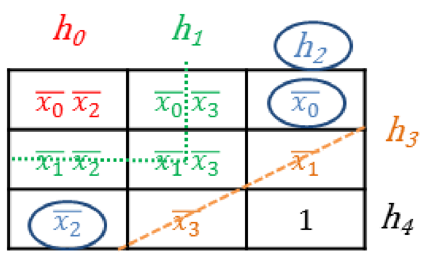
$$h_1 = f_0 g_1 + f_1 g_0 + f_1 g_1 = \overline{x_0}\,\overline{x_3} + \overline{x_1}\,\overline{x_2} + \overline{x_1}\,\overline{x_3} \tag{4.2.19c}$$

$$h_2 = f_0 + g_0 = \overline{x_0} + \overline{x_2} \tag{4.2.19d}$$

$$h_3 = f_1 + g_1 = \overline{x_1} + \overline{x_3} \tag{4.2.19e}$$

where the level $h_4 = 1$ was removed. Alternatively, we can draw the matrix and obtain the levels, as illustrated in Fig. 4.2.

Figure 4.2 – Matrix view for *Example 4.2.2.*



h_0
h_1
h_2
$\overline{x_0}\,\overline{x_2}$
$\overline{x_0}\,\overline{x_3}$
$\overline{x_0}$
h_3
$\overline{x_1}\,\overline{x_2}$
$\overline{x_1}\,\overline{x_3}$
$\overline{x_1}$
$\overline{x_2}$
$\overline{x_3}$
1
h_4

Source: The author.

In the following, we demonstrate that the proposed method is correct. Initially, we show that the method is correct when only the initial terms from (4.2.9) are considered, this is the base case. Then, we show that if the sum of the first i terms in (4.2.9) is equivalent to the sum of the first j terms in the resulting RBF, obtained from (4.2.11) and (4.2.12), then the sum of the first $i+1$ terms in (4.2.9) is equivalent to the RBF given by (4.2.11) and (4.2.12), this is the induction step.

Base case : We can rewrite (4.2.9) as follows:

$$h^* = h_0^* + h_1^* + \cdots + h_{|f|+|g|}^* \tag{4.2.20}$$

where each h_i^* in (4.2.20) corresponds to the sum of all terms in (4.2.9) containing *i* negative literals, as follows:

$$h_0^* = f_0 g_0 \tag{4.2.21a}$$

$$h_1^* = f_0 \overline{g_1} g_2 + \overline{f_1} f_2 g_0 \tag{4.2.21b}$$

$$h_2^* = f_0 \overline{g_1}\,\overline{g_3} g_4 + \overline{f_1} f_2\,\overline{g_1} g_2 + \overline{f_1}\,\overline{f_3} f_4 g_0 \tag{4.2.21c}$$

$$\begin{aligned} h_i^* = f_0 \overline{g_1}\,\overline{g_3} \dots \overline{g_{2\iota-1}} g_{2i} + \overline{f_1} f_2\,\overline{g_1} \dots \overline{g_{2\iota-3}} g_{2i-2} + \cdots + \overline{f_1} \dots \overline{f_{2\iota-3}} f_{2i-2} \overline{g_1} g_2 \\ + \overline{f_1}\,\overline{f_3} \dots \overline{f_{2\iota-1}} f_{2i} g_0 \end{aligned} \tag{4.2.21d}$$

For the base case, we consider only the two first terms of (4.2.21), denoted by $h_{0:1}^*$, as follows:

$$h_{0:1}^* = f_0 g_0 + \overline{f_1} f_2 g_0 + f_0 g_0 \overline{g_1} \tag{4.2.22}$$

Equation (4.2.22) equals the first terms of the RBF given by (4.2.11) and (4.2.12), as follows:

$$h_{0:2} = h_0 + \overline{h_1} h_2 \tag{4.2.23}$$

By replacing the values for h_0, h_1 and h_2 in (4.2.23), we obtain:

$$\begin{gathered} h_{0:2} = f_0 g_0 + \overline{(f_1 g_0 + f_0 g_1 + f_1 g_1)} (f_2 g_0 + f_0 g_2) = \\ f_0 g_0 + \overline{f_0}\,\overline{f_1} f_2 g_0 + f_0 \overline{g_0}\, g_1 g_2 = f_0 g_0 + \overline{f_1} f_2 g_0 + f_0\,\overline{g_1} g_2 \end{gathered} \tag{4.2.24}$$

Since (4.2.22) and (4.2.24) are equivalent, the base case holds.

Inductive step: The inductive hypothesis is that the sum of the first i terms in (4.2.20) (denoted by $h^*_{0:i}$) is equivalent to the RBF obtained from (4.2.11) and (4.2.12) truncated at level h_{2i} (denoted by $h_{0:2i}$), as follows:

$$h^*_{0:i} = h_{0-2i} \tag{4.2.25}$$

where $h^*_{0:i}$ and $h_{0:2i}$ are given, respectively, by:

$$h^*_{0:i} = h_0{}^* + h_1{}^* + \cdots + h_i{}^* \tag{4.2.26}$$

and

$$h_{0:2i} = h_0 + \overline{h_1}h_2 + \cdots + \overline{h_1} \ldots \overline{h_{2\imath-1}}h_{2i} \tag{4.2.27}$$

We show that if (4.2.25) holds, then the sum of the first $i+1$ terms in (4.2.20), denoted by $\mathrm{h}^*_{0:\mathrm{i}+1}$, is equivalent to RBF obtained from (4.2.11) and (4.2.12) truncated at level h_{2i+2}, denoted by $h_{0:2i+2}$, as given by:

$$h^*_{0:i+1} = h_{0:2i+2} \tag{4.2.28}$$

where the expressions $h^*_{0:i+1}$ and $h_{0:2i+2}$ are given by:

$$h^*_{0:i+1} = h^*_{0:1} + h^*_{i+1} \tag{4.2.29}$$

and

$$h_{0:2i+2} = h_{0:2i} + \overline{h_1} \ldots \overline{h_{2\imath+1}}h_{2i+2} \tag{4.2.30}$$

From (4.2.21), h^*_{i+1} is defined in a manner such that, for any term in h^*_{i+1}, summing the indexes of the positive literals yields *2i+2*. From (4.2.11), the same analysis is valid for h_{2i+2} . Therefore, for each term τ in h^*_{i+1}, there is a term ψ in h_{2i+2} such that τ and ψ have the same positive literals. We want to check if, for all such pairs (τ,ψ), the following relationship holds:

$$h^*_{0:i} + \tau = h_{0:2i} + \overline{h_1}\,\overline{h_3}\,\overline{h_{2\iota+1}}\Psi \tag{4.2.31}$$

In the following, we analyze the case for a single pair (τ, Ψ). The analysis for the other cases is similar. Let τ and Ψ be given, respectively, by:

$$\tau = f_0\overline{g_1}\,\overline{g_3}\dots\overline{g_{2\iota+1}}\,g_{2i+2} \tag{4.2.32}$$

And:

$$\Psi = f_0\, g_{2i+2} \tag{4.2.33}$$

By replacing (4.2.32) and (4.2.33) into (4.2.31) yields:

$$h^*_{0:i} + f_0\overline{g_1}\,\overline{g_3}\dots\overline{g_{2\iota+1}}\,g_{2i+2} = h_{0-2i} + \overline{h_1}\,\overline{h_3}\,\overline{h_{2\iota+1}}f_0\, g_{2i+2} \tag{4.2.34}$$

To show that (4.2.34) holds, we make the following observations:

1) If any of f_0 and g_{2i+1} is 0, then both (4.2.32) and (4.2.33) are 0 and (4.2.34) holds.

2) If both f_0 and g_{2i+1} are 1 and there is a q ∈ {1,3,5,..,2i-1} such that $g_q = 1$, then both $f_0\overline{g_1}\,\overline{g_3}\dots\overline{g_{2\iota+1}}\,g_{2i+2}$ and $\overline{h_1}\,\overline{h_3}\,\overline{h_{2\iota+1}}f_0\, g_{2i+2}$ are equal to 0. Notice that in each h_i with odd *i* there is a term of the form $f_0 g_i$. Therefore, (4.2.34) is reduced to (4.2.25) and, due to the inductive hypothesis, (4.2.34) holds.

3) If all $g_q = 0$ for all q ∈ {1,3,5,..,2i-1}, and there is a $g_k = 1$, where k ∈ {2,4,6,..,2i}, then both $h^*_{0:i}$ and $h_{0:2i}$ are 1. This happens because for any g_k with even *k* there is a term in $h^*_{k/2}$ given by $f_0\overline{g_1}\,\overline{g_3}\dots\overline{g_{k-1}}g_k$.

4) The last case left to consider is $f_0 = g_{2i+2} = 1$ and $g_q = 0$, being $1 \le q \le 2i$. In this case, (4.2.34) is reduced to the following:

$$\overline{g_{2\iota+1}} = \overline{h_{2\iota+2}} \tag{4.2.35}$$

We show that, under these conditions, $h_{2i+1} = g_{2i+1}$. The expression for h_{2i+1} is given by (4.2.12d). Since $f_0 = 1$ and all $g_q = 0$ for q in {1,2,3,..,2i}, only the last two terms in (4.2.12d) are not equal to 0. Therefore, h_{2i+1} becomes:

$$h_{2i+1} = g_{21+1} + f_1 g_{2i+1} = g_{2i+1} \tag{4.2.36}$$

Since (4.2.35) holds, we can conclude that if the RBF for the first h_i^* terms is correct, then the RBF including the term h_{i+1}^* is also correct. Since the base case guarantees that the RBF considering only terms h_0^* and h_1^* is correct, the final RBF is also correct.

4.2.2.2 Sharp, NOR and OR decompositions

The method can also be applied to negative phase RBF. Let $h = f \cdot \bar{g}$ ($f \cdot \bar{g}$ is also known as the sharp operation (BRAYTON, 1984)). RBF $\bar{g}$ can be written as follows:

$$\bar{g} = \overline{g_0}g_1 + \overline{g_0}\,\overline{g_2}g_3 + \cdots + \overline{g_0}\,\overline{g_2}\,...\,\overline{g_{|g|-1}} \tag{4.2.37}$$

A Boolean expression $h^*$for h is the following:

$$h^* = \overline{g_0}\,f\,g_{\backslash g_0} = \overline{g_0}\,h' \tag{4.2.38}$$

where $h' = f\,g_{\backslash g_0}$, and $g_{\backslash g_0}$ is obtained by removing the term $\overline{g_0}$ from $\bar{g}$, as follows:

$$g_{\backslash g_0} = g_1 + \overline{g_2}g_3 + \cdots + \overline{g_2}\,...\,\overline{g_{|g|-1}} \tag{4.2.39}$$

Notice that (4.2.38) is a positive phase RBF. Therefore, an RBF for h' is obtained through the AND decomposition. The RBF for h is a negative phase RBF where the levels are given by:

$$h_0 = g_0 \tag{4.2.40a}$$

$$h_{i+1} = h_i' \tag{4.2.40b}$$

In summary, g_0 equals h_0. The other levels of h are obtained through the AND decomposition between f and $g_{\backslash g_0}$.

In the following, we discuss the case when h is an h as an AND decomposition of two negative phase RBF, such as $h = \bar{f} \cdot \bar{g}$. This case also applies to consider an OR decomposition where $h = f + g$. Similarly to (4.2.38), we write an expression h^* for h as follows:

$$h^{*} = \overline{f_0}\,\overline{g_0}\,f_{\backslash f_0}\,g_{\backslash g_0} = \overline{f_0}\overline{g_0}\,h' \tag{4.2.41}$$

where $h' = f_{\backslash f_0}\,g_{\backslash g_0}$, being $f_{\backslash f_0}$ and $g_{\backslash g_0}$ obtained by removing the term $\overline{f_0}$ from f and the term $\overline{g_0}$ from $g,$ respectively, resulting in an expression similar to (4.2.38). The RBF h' is obtained through the AND decomposition between $f_{\backslash f_0}$ and $g_{\backslash g_0}$. The resulting negative phase RBF h is the following:

$$h = h_0 + \overline{h_1}h_2 + \cdots \tag{4.2.42}$$

where

$$h_0 = f_0 + g_0 \tag{4.2.43a}$$

$$h_{i+1} = h_i' \tag{4.2.43b}$$

4.2.2.3 XOR and XNOR decompositions

By combining the previous strategies, we can derive a method to consider XNOR decompositions.

$$h = \overline{f \oplus g} = fg + \bar{f}\,\bar{g} = t + u \tag{4.2.44a}$$

$$t = fg \tag{4.2.44b}$$

$$u = \bar{f}\bar{g} \tag{4.2.44c}$$

where $\oplus$ is the XOR operator. A positive phase RBF h can be obtained through the already described decompositions. The levels for RBF h are the following:

$$h_0 = f_0 g_0 \tag{4.2.45a}$$

$$h_1 = f_1 g_0 + f_0 g_1 \tag{4.2.45b}$$

$$h_2 = f_2 g_0 + f_0 g_2 + f_1 g_1 \tag{4.2.45c}$$

$$h_{2i-1} = f_{2i-1}g_0 + f_{2i-2}g_1 + f_{2i-3}g_2 + f_{2i-4}g_3 + + \cdots + f_3 g_{2i-4} + f_2 g_{2i-3} + f_1 g_{2i-2} + f_0 g_{2i-1} \tag{4.2.45d}$$

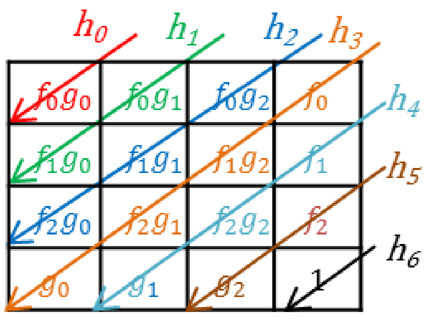
$$h_{2i} = f_{2i}g_0 + f_{2i-2}g_2 + \cdots + f_2 g_{2i-2} + f_0 g_{2i} + f_{2i-1}g_1 + f_{2i-3}g_3 + \cdots + f_1 g_{2i-1} \tag{4.2.46e}$$

The XNOR decomposition is similar to the AND decomposition. However, the term $f_i g_i$ is inserted at h_{2i} instead of h_{2i-1}. Therefore, each level h_k is composed by the sum of all $f_i g_j$, which satisfy:

$$i + j = k \qquad (4.2.47)$$

Notice that (4.2.47) is valid for both even and odd levels. The matrix view for XNOR decompositions is illustrated in Fig. 4.3, which shows that the levels can be defined by drawing diagonal lines on the matrix.

Figure 4.3 – Matrix view for XNOR decompositions.



Source: The author.

Example 4.2.3: Consider an expression h^* given by:

$$h^* = (\overline{x_0}x_1) \oplus (\overline{x_2}x_3) \qquad (4.2.48)$$

Expression h^* is the XOR between two negative phase RBF $\bar{f}$ and $\bar{g}$ which are given by the following expressions:

$$\bar{f} = f_0 + \overline{f_1} \qquad (4.2.49a)$$
$$f_0 = \overline{x_1} \qquad (4.2.49b)$$
$$f_1 = \overline{x_0} \qquad (4.2.49c)$$

and

$$\bar{g} = g_0 + \overline{g_1} \qquad (4.2.50a)$$
$$g_0 = \overline{x_3} \qquad (4.2.50b)$$
$$g_1 = \overline{x_2} \qquad (4.2.50c)$$

By replacing the values of f and g into (4.2.48), an expression for h^* is obtained:

$$h^{*} = (x_0 + \overline{x_1}) \oplus (x_2 + \overline{x_3}) \tag{4.2.51}$$

By applying (4.2.47) to the products $f_i g_j$ obtained from (4.2.49) and (4.2.50), we obtain the resulting negative phase RBF $\bar{h}$. Notice that the phase is negative because the top operator is an XOR. RBF $\bar{h}$ is given by the following:

$$\bar{h} = h_0 + \overline{h_1} h_2 \tag{4.2.51a}$$

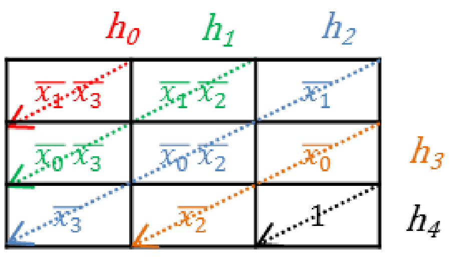
$$h_0 = \overline{x_1}\,\overline{x_3} \tag{4.2.51b}$$

$$h_0 = \overline{x_1}\,\overline{x_2} + \overline{x_0}\,\overline{x_3} \tag{4.2.51c}$$

$$h_2 = \overline{x_1} + \overline{x_3} + \overline{x_0}\,\overline{x_2} \tag{4.2.51d}$$

Alternatively, we can draw the corresponding matrix and the corresponding diagonal lines, as shown in Fig. 4.4. As expected, the same result given by (4.2.51) is obtained.

Figure 4.4 – Matrix view for Example 4.2.3.


h_0
h_1
h_2
$\overline{x_1}\,\overline{x_3}$
$\overline{x_1}\,\overline{x_2}$
$\overline{x_1}$
$\overline{x_0}\,\overline{x_3}$
$\overline{x_0}\,\overline{x_2}$
$\overline{x_0}$
h_3
$\overline{x_3}$
$\overline{x_2}$
1
h_4

Source: The author.

4.3 SIMPLIFICATION OF RBF

In this section, we discuss methods to simplify an RBF by either removing a cube from a certain level or by moving a cube to another level. Clearly, a cube that is dominated by another cube at the same level h_i can be removed from the RBF. Since all h_i are negative unate, a single cube containment (SCC) algorithm can be used to identify such cubes (BRAYTON, 1984).

A cube c_1, in a level h_i, is also redundant if it is dominated by a cube c_2 in a level h_j with $j \leq i$. The rationale for this is the following. Since the evaluation of h_i is only important when all levels h_k, with $k < i$, are equal to 0, if c_1 is dominated by c_2, then c_1 can only be 1 when some level h_k is 1. However, in this case, the value of c_1 does not matter.

Two different possibilities to move a cube from h_i to either h_{i+2} or h_{i-2} are presented. After the cube is moved, it can either become redundant or make another cube redundant. Moreover, if all cubes from h_i are moved, such that h_i becomes empty, we can merge the levels h_{i-1} and h_{i+1}.

Case 1: A cube c_i can be moved from level h_j to h_{j+2} if $c_i = 1$ leads to $h_{j+1} \subseteq h_{j-1}$.

To illustrate this simplification, consider an RBF f as follows:

$$f = f_0 + \overline{f_1} f_2 + \overline{f_1}\,\overline{f_3} f_4 \tag{4.3.1a}$$

$$f = f_0 + \overline{f_1}(c_0 + \cdots + c_i + \cdots + c_k) + \overline{f_1}\,\overline{f_3} f_4 \tag{4.3.1b}$$

If the cube c_i in f_2 is moved to f_4, the resulting expression is the following:

$$f = f_0 + \overline{f_1}(c_0 + \cdots + c_k) + \overline{f_1}\,\overline{f_3}(f_4 + c_i) \tag{4.3.2}$$

Equations (4.3.1b) and (4.3.2) can only differ when $f_0 = f_1 = 0$ and $c_2 = f_3 = 1$. However, if $c_i = 1$ leads to $f_3 \subseteq f_1$, then such a condition cannot occur and the equations are equivalent.

Case2: A cube c_i can be moved from level h_j to h_{j-2} if $c_i = 1$ leads to $h_{j-1} \subseteq h_{j-2}$.

Consider the same RBF as in (4.3.1). When the cube c_i in f_2 is moved to f_0, the resulting expression is the following:

$$f = f_0 + c_i + \overline{f_1}(c_0 + \cdots + c_k) + \overline{f_1}\,\overline{f_3} f_4 \tag{4.3.3}$$

Equations (4.3.1b) and (4.3.3) can only differ when $f_0 = 0$ and $f_1 = 1$. However, if $c_i = 1$ leads to $f_0 \subseteq f_1$, then such a condition never occurs and the equations are equivalent.

Example 4.3.1: Consider an RBF given as follows:

$$f = f_0 + \overline{f_1} f_2 + \overline{f_1}\,\overline{f_3} f_4 \tag{4.3.4a}$$

$$f_0 = \overline{x_0}\,\overline{x_1}\,\overline{x_2} \tag{4.3.4b}$$

$$f_1 = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} \tag{4.3.4c}$$

$$f_2 = \overline{x_0} \tag{4.3.4d}$$

$$f_3 = \overline{x_1}\,\overline{x_2} \tag{4.3.4e}$$

$$f_4 = \overline{x_1} + \overline{x_2} \tag{4.3.4f}$$

Consider the cube $\overline{x_1}\,\overline{x_2}$ in f_3. When $\overline{x_1}\,\overline{x_2} = 1$, f_1 becomes f_1', as follows:

$$f_1{}' = \overline{x_0} \tag{4.3.5}$$

Since $f_1' = f_2$, the condition $h_{j-1} \subseteq h_{j-2}$ of the Case 2 is satisfied. Therefore, we can move the cube $\overline{x_1}\,\overline{x_2}$ from f_3 to f_1. The resulting expression is the following:

$$f = f_0 + \overline{f_1} f_2 \tag{4.3.6a}$$

$$f_0 = \overline{x_0}\,\overline{x_1}\,\overline{x_2} \tag{4.3.6b}$$

$$f_1 = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} + \overline{x_1}\,\overline{x_2} \tag{4.3.6c}$$

$$f_2 = \overline{x_0} + \overline{x_1} + \overline{x_2} \tag{4.3.6d}$$

Equation (4.3.6) is the standard RBF for the XNOR3. Also notice that, since $\overline{x_1}\,\overline{x_2}$ was the only cube in f_3, the levels f_4 and f_2 are merged.

The proposed simplifications can improve the quality of RBF in several cases. However, there are solutions which cannot be improved through the proposed methods. For instance, the methods described in (RAGHUVANSHI, 2014) and in (TEODOROVIC, 2013) provide the following RBF for the 2-to-1 multiplexer, for which the truth table is shown in Table 4.4:

$$f = \overline{f_0} f_1 + \overline{f_0}\,\overline{f_2} \tag{4.3.7a}$$

$$f_0 = \overline{x_1}\,\overline{x_2} + \overline{x_0}\,\overline{x_2} \tag{4.3.7b}$$

$$f_1 = \overline{x_0} \tag{4.3.7c}$$

$$f_2 = \overline{x_1} \tag{4.3.7d}$$

Using the cases previously described, no cube can be moved or removed. However, the cube $\overline{x_1}\,\overline{x_2}$ can be removed from (4.3.7) without modifying the described function. Defining an efficient method to identify these cases remains an open problem.

4.4 EXPERIMENTAL RESULTS

We begin by evaluating all functions with at most four inputs. We use the methods described in (RAGHUVANSHI, 2014), which is named COVER method, and in (TEODOROVIC, 2013), which is named PARTITION method, as references. We evaluate both the results obtained from the algorithm described in the corresponding work as well as the results obtained by applying our optimization algorithm over the initial results. All results consider our own implementation of the methods. In Table 4.5, it is presented the results for the different methods considering the average and the worst case scenarios. We observed that the solutions provided by the COVER and the PATITION methods are the same. Hence, they are grouped into the same line in Table 4.5. It can be seen that the methods have similar solution quality for these simple functions. For all cases, the worst cases are the XOR4 and XNOR4 functions which require 16 cycles.

Table 4.5 – Comparison of RBF synthesis methods for 4-input functions.

Method	Average	Worst case
COVER/PARTITION (Original)	8.24	16
COVER/PARTITION (Optimized)	7.84	16
DEC-RBF	7.78	16
FC-RBF	7.49	16

Source: The author.

The second set of experiments considers more complex functions for which we cannot use the FC-RBF method. These functions have been taken from the ESPRESSO book literature (BRAYTON, 1984). The main characteristics of these functions are shown in Table 4.6, which also presents the design name, the corresponding output index, the total number of inputs of the design, the number of care inputs for the target output, and the number of cubes for the target output. The data in Table 4.6 serves as reference for other experiments performed herein.

Due to the limitations on the representation format used by the COVER and PARTITION methods, only functions with at most 20 inputs have been taken into account. Results are summarized in Table 4.7 for the 15 functions that the COVER method yields the largest RBF. The first column depicts the design name and, inside the parenthesis, the output index of the design. The same results for the COVER and PARTITION methods are obtained.

When compared to previous approaches, the proposed DEC-RBF method reduces the average number of cycles by 30% when considering the 15 largest RBF. In turn, the average runtime is reduced by 95%. The runtime improvement is a direct consequence of the data structure. For instance, output (5) of design ‘table5’ is a 17-input function which has an ISOP with 74 cubes. Therefore, the input size for the COVER method is 2^{17}, whereas the input size for DEC-RBF is 1258. Nevertheless, for some cases such as output (7) of ‘alu4’, the COVER method is significantly faster than DEC-RBF. Moreover, by applying the proposed optimization procedure to the COVER method reduces the average number of cycles by about 10% while having negligible impact on the runtime.

Table 4.6 – Main characteristics of SOP used to evaluate the methods.

Design	Output index	Inputs	Care inputs	Cubes		Design	Output index	Inputs	Care inputs	Cubes
alu4	2	14	12	50		table3	0	14	14	51
	4	14	14	181			4	14	14	70
	7	14	14	182		table5	1	17	17	41
apex1	22	45	39	79			3	17	17	54
	34	45	28	37			5	17	17	74
apex2	0	39	36	278			6	17	17	55
	2	39	35	523			10	17	17	21
bca	31	26	16	20			11	17	17	61
bcb	16	26	15	22			13	17	17	71
	30	26	16	20			14	17	17	55
bcd	33	26	15	7		test2	0	11	11	164
cordic	0	23	23	143			8	11	11	171
	1	23	23	771			14	11	11	170
intb	1	15	12	50			14	11	11	160
	6	15	15	230			15	11	11	163
max1024	5	10	10	121			18	11	11	158
max512	5	9	9	62			19	11	11	160
misex3	7	14	14	141			22	11	11	155
	9	14	14	113			24	11	11	177
	13	14	14	116			29	11	11	163
prom1	13	9	9	55		ti	4	47	21	27
	15	9	9	55		vg2	4	25	16	30
	20	9	9	52			5	25	18	40
rd84	1	8	8	128		vtx1	1	27	18	40
seq	5	41	38	105		x1dn	1	27	18	40
signet	0	39	28	46		x6dn	1	39	34	34
	1	39	32	39		x9dn	2	27	18	40
	2	39	28	44		xparc	37	41	30	60
t481	0	16	16	481						

Source: The author.

Table 4.7 – Comparison among different RBF synthesis methods for functions with at most 20 inputs taken from ESPRESSO literature (BRAYTON, 1984).

Design (out)	DEC-RBF		COVER				PARTITION			
			Original		Optimized		Original		Optimized	
	Cycles	Runtime (s)	Cycles	Runtime (s)	Cubes	Runtime (s)	Cycles	Runtime (s)	Cubes	Runtime (s)
alu4 (7)	1094	11.94	1275	2.25	1191	2.36	1275	12.40	1191	12.50
alu4 (4)	761	3.68	994	1.79	872	1.84	994	12.10	872	12.20
misex3c (13)	717	2.27	990	1.50	822	1.58	990	11.90	822	12.00
table5 (5)	448	0.65	928	85.40	859	85.40	928	926.90	859	926.90
table5 (13)	314	0.38	631	80.50	571	80.50	631	918.80	571	918.80
misex3 (7)	435	1.58	618	1.17	527	1.23	618	11.300	527	11.30
table3 (4)	448	0.44	603	1.22	558	1.25	603	12.00	558	12.00
table5 (6)	397	0.3	603	64.50	543	64.50	603	911.80	543	911.60
table5 (14)	392	0.42	517	57.40	495	57.40	517	898.10	495	899.40
misex3 (3)	303	1.56	513	1.00	405	1.02	513	11.20	405	11.20
table5 (11)	394	0.44	507	74.40	437	74.40	507	924.10	437	923.00
misex3 (9)	443	1.03	499	0.93	439	0.96	499	11.20	439	11.20
table3 (0)	334	0.15	485	1.10	421	1.10	485	11.70	421	11.70
table5 (3)	267	0.19	484	69.20	454	69.20	484	928.30	454	928.20
table5 (1)	323	0.16	472	66.10	433	66.20	472	909.60	433	909.10
AVG.	471	1.67	674	33.90	601	33.90	674	434.10	601	434.10

Source: The author.

5 SYNTHESIS AND EVALUATION OF SUM-OF-RECURSIVE-BOOLEAN-FORMS

In this chapter, we propose the use of diverse multilevel forms in RSD-IMP logic structure. In Section 5.1, we propose the use of sum-of-RBF (SRBF). A SRBF is given by:

$$f = f_1 + \cdots + f_\phi \tag{5.1}$$

where each f_i is an RBF. We also investigate the consequences of setting an upper bound m on the number of levels of each RBF f_i in (5.1). In particular, when $m = 2$, the resulting expression is a four-level form (FLF), that can be written as follows:

$$f = fn_1\overline{fp_1} + \cdots + fn_\beta\overline{fp_\beta} = \tau_1 + \cdots + \tau_\beta \tag{5.2}$$

where all fp_i and fn_i are SOPs containing only negative cubes. Each term τ_i is defined as follows:

$$\tau_i = fn_1\overline{fp_1} \tag{5.3}$$

Notice that each term τ_i is a negative phase RBF comprising two levels. The sequence of operations to evaluate an SRBF, with two work RSDs, is directly obtained from the operations to evaluate each RBF f_i individually.

In Section 5.2, we consider a class of expressions named single-cube factor RBF (SC-FSRBF) which are written in the following form:

$$f = \pi_1 f_1 + \cdots + \pi_m f_m \tag{5.4}$$

where each π_i is a positive cube and each f_i is an SRBF or an SC-FSRBF. SC-FSRBF requires extra RSDs to be evaluated. In Section 5.2, we show that the number of RSDs is easily derived from the expression such that this information can be included into a logic synthesis algorithm.

5.1 SUM-OF-RECURSIVE-BOOLEAN-FORMS

We propose an SOP based method to obtain SRBF. The goal is to minimize the number of cycles to evaluate the resulting SRBF, which is directly related to the number of cubes in SRBF. Any SOP can be directly written as an SRBF. More specifically, an SOP can be transformed into an FLF in linear time with respect to the number of literals in the SOP. Consider a binate cube c with η negative literals and ρ positive literals, as follows:

$$c = \overline{xn_1} \dots \overline{xn_\eta} xp_1 \dots xp_\rho \tag{5.1.1}$$

The cube c can be written in the form of (5.1), as follows:

$$fn = \overline{xn_1} \dots \overline{xn_\eta} \tag{5.1.2a}$$

$$fp = \overline{xp_1} + \dots + \overline{xp_\rho} \tag{5.1.2b}$$

Positive and negative cubes are written by making $fn_\imath = 1$ and $\overline{fp_\imath} = 1$, respectively. Notice that each positive literal in the SOP becomes a negative cube with a single literal in the SRBF. A cube with $\rho \geq 1$ positive literals requires $2 + \rho$ cycles to be evaluated.

We propose a heuristic greedy algorithm to synthesize SRBF from SOP. The proposed algorithm aims to group a set of cubes $\{c_{i1}, c_{i2}, \dots, c_{ik}\}$ of the SOP into a single term τ_i. Then, an RBF is created for each τ_i. In this manner, the number of cubes in the SRBF can be reduced when compared to the original FLF obtained from the initial SOP representation.

The first step to obtain an SRBF is to evaluate which combinations of two cubes lead to improvements. This analysis generates a graph where each vertex represents a cube and edges connect cubes that improve the quality solution when combined. The edge weight represents how much the quality solution is improved. The graph generation algorithm is depicted in Algorithm 5.1.1. Each cube is transformed into a term τ_i, given by (5.1.2), and each τ_i becomes a vertex in the graph. For each pair of terms τ_i and τ_j, the algorithm generates an RBF for $\tau_{i,j} = \tau_i + \tau_j$, using the method described in Chapter 4. Let $|\tau_y|$ represent the number of cubes in some term τ_y, and $\omega_{i,j} = |\tau_i| + |\tau_j| - |\tau_{i,j}|$. If $\omega_{i,j} \geq 0$, then an edge $e_{i,j}$ with weight $\omega_{i,j}$ connecting τ_i and τ_j is created.

Algorithm 5.1.1 – Graph generation algorithm.

```
    Input: SOP $f$
    Output: Graph depicting the possible two cube combinations
1.  transform each cube $c_i$ into a $\tau_i$ given by (5.1.2)
2.  for each pair $\tau_i$, $\tau_j$
3.      create a RBF for $\tau_{i,j} = \tau_i + \tau_j$
5.      $\omega_{i,j} = 1 + |\tau_i| + |\tau_j| - |\tau_{i,j}|$
6.      if $\omega_{i,j} \geq 0$, then create edge $e_{i,j}$ with weight $\omega_{i,j}$
7.  end for
```

The second step of the algorithm is to merge groups given the graph obtained from Algorithm 5.1.1. We consider three different variations for this step which are described in Algorithm 5.1.2, in Algorithm 5.1.3 and in Algorithm 5.1.4. Algorithm 5.1.2 aims on the best possible solution. On the other hand, Algorithm 5.1.3 and Algorithm 5.1.4 explore a trade-off between the solution quality and execution time.

Algorithm 5.1.2 – SOP Based Synthesis of SRBF.

```
    Input: SOP $f$
    Output: SRBF
1.  Create graph using Algorithm 5.1.1
2.  sort edges from the heaviest to the lightest
3.  while there is an unvisited edge $e_{i,j}$
4.    $\tau_{i,j} = \tau_i \cup \tau_j$
5.    replace $\tau_i$ and $\tau_j$ by $\tau_{i,j}$
6.    create edges for $\tau_{i,j}$ (as in lines Algorithm 5.1.1)
7.  end for
8.  return all $\tau$
```

After the graph is created, Algorithm 5.1.2 sorts the edges from the heaviest to the lightest and selects the heaviest edge $e_{i,j}$, which connects τ_i and τ_j. Then, all edges from τ_i and τ_j are removed. τ_i and τ_j are replaced by $\tau_{i,j}$ and the edges for $\tau_{i,j}$ are created, as described previously. After all edges are processed, the execution of the algorithm stops.

Example 5.1.1: Consider the function given by the following ISOP:

$$f = \overline{x_1}\,\overline{x_2}\,x_3x_4x_5 + \overline{x_1}\,\overline{x_2}x_6x_7x_8 + x_1\overline{x_3}\,\overline{x_9} + x_1\overline{x_6}\,\overline{x_7} \tag{5.1.3}$$

If f is directly transformed into a FLF, 16 cycles are used. In the following, let:

$$c_1 = \overline{x_1}\,\overline{x_2}\,x_3x_4x_5 \tag{5.1.4a}$$

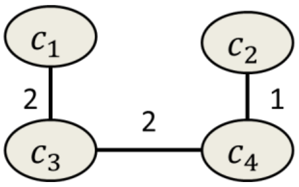
$$c_2 = \overline{x_1}\,\overline{x_2}x_6x_7x_8 \tag{5.1.4b}$$

$$c_3 = x_1\overline{x_3}\,\overline{x_9} \tag{5.1.4c}$$

$$c_4 = x_1\overline{x_6}\,\overline{x_7} \tag{5.1.4d}$$

For each pair of cubes in the SOP, we obtain an RBF representing the sum of such cubes. If the number of cycles to evaluate the cubes as an RBF is smaller than the initial number of cycles, then an edge is created connecting the corresponding vertices. Fig. 5.1 depicts the resulting graph. It can be seen that merging c_1 and c_3 reduces the number of cycles by two. In turn, merging c_1 with either c_2 or c_4 increases the number of cycles.

Figure 5.1 - Graph for Example 5.1.1.



Source: The author.

The first edge to visited can be either $e_{1,3}$ or $e_{3,4}$. For sake of simplicity, we assume that $e_{1,3}$ is the first edge. When this edge is visited, the cubes c_1 and c_3 are combined into a group τ_5. The expression for τ_5 is the following:

$$\tau_5 = \overline{\tau_{5_0}}\tau_{5_1} \qquad (5.1.5a)$$

$$\tau_{5_0} = \overline{x_1}\,\overline{x_3} + \overline{x_1}\,\overline{x_4} + \overline{x_1}\,\overline{x_5} \qquad (5.1.5b)$$

$$\tau_{5_1} = \overline{x_1}\,\overline{x_2} + \overline{x_3 x_9} \qquad (5.1.5b)$$

The next step is to create the edges for τ_5. The RBF to compute $\tau_5 + c_2$, comprises 15 cubes, being a worse solution than evaluating τ_5 and c_2 individually. Therefore, no edge exists between τ_5 and c_2. On the other hand, the RBF for $\tau_5 + c_4$ comprises eight cubes. In this case, using such an RBF yields the same number of cycles when compared to the evaluation of τ_5 and c_4 independently.

The next visited edge is $e_{2,4}$ which has a weight of 1. Therefore, the algorithm merges c_2 and c_4 into τ_6. The RBF τ_6 is given by the following:

$$\tau_6 = \overline{\tau_{6_0}}\tau_{6_1} \qquad (5.1.6a)$$

$$\tau_{5_0} = \overline{x_1}\,\overline{x_6} + \overline{x_1}\,\overline{x_7} + \overline{x_1}\,\overline{x_8} \qquad (5.1.6b)$$

$$\tau_{5_1} = \overline{x_1}\,\overline{x_2} + \overline{x_6}\,\overline{x_7} \qquad (5.1.6b)$$

If τ_5 and τ_6 are combined into a single RBF, the resulting expression comprises 14 cubes, being a worse solution than evaluating τ_5 and τ_6 individually. Therefore, the final SRBF is as follows:

$$f = \tau_5 + \tau_6 \tag{5.1.7}$$

where τ_5 and τ_6 are given by (5.1.5) and (5.1.6), respectively. The evaluation of the FLF requires 12 cycles. The sequence of operations is shown in Table 5.1. Therefore, a reduction of four with respect to the original solution has been obtained.

Table 5.1 – Sequence of operations to evaluate (5.1.6).

		y_0	y_1
1.	$y_1 = 0, y_0 = 0$	0	0
2.	$(x_1 + x_3) \rightarrow y_1$		$\overline{x_1}\,\overline{x_3}$
3.	$(x_1 + x_4) \rightarrow y_1$		$\overline{x_1}\,\overline{x_3} + \overline{x_1}\,\overline{x_4}$
4.	$(x_1 + x_5) \rightarrow y_1$		$\overline{x_1}\,\overline{x_3} + \overline{x_1}\,\overline{x_4} + \overline{x_1}\,\overline{x_5}$
5.	$(x_1 + x_2 + y_1) \rightarrow y_0$	$\overline{x_1}\,\overline{x_2}\,x_3x_4x_5$	
6.	$(x_3 + x_9 + y_1) \rightarrow y_0$	τ_5	
7.	$y_1 = 0$		0
8.	$(x_1 + x_6) \rightarrow y_1$		$\overline{x_1}\,\overline{x_6}$
9.	$(x_1 + x_7) \rightarrow y_1$		$\overline{x_1}\,\overline{x_6} + \overline{x_1}\,\overline{x_7}$
10.	$(x_1 + x_8) \rightarrow y_1$		$\overline{x_1}\,\overline{x_6} + \overline{x_1}\,\overline{x_7} + \overline{x_1}\,\overline{x_8}$
11.	$(x_1 + x_2 + y_1) \rightarrow y_0$	$\tau_5 + \overline{x_1}\,\overline{x_2}x_6x_7x_8$	
12.	$(x_6 + x_7 + y_1) \rightarrow y_0$	$\tau_5 + \tau_6$	

Source: The author.

In Algorithm 5.1.2, when that two vertices are merged into a τ_i, new RBFs are generated for all possible pairs involving the new term τ_i. However, most of these RBFs are not used in the final solution. Hence, the execution time can be improved if only the RBFs that are used in the final solution are generated. However, it is only possible to know which RBFs are needed after generating and testing them. In Algorithm 5.1.3, we consider a different approach to generate the edges for the new created terms in order the trade-off runtime for solution quality. We modify the procedure used to create the edges for a node τ_k resulting from the combination of groups τ_i and τ_j. Instead of reevaluating all vertices to compute new edges, τ_k has only edges $e_{k,m}$ for nodes τ_m that have a connection to both τ_i and τ_j. The weight of $e_{k,m}$ corresponds to the largest one between $\omega_{i,m}$ and $\omega_{j,k}$. In this sense, runtime is reduced because RBFs are only generated when creating the graph and after all groupings have been performed.

Algorithm 5.1.3 – Clique-based synthesis of SRBF.

```
     Input: SOP $f$
     Output: SRBF
1.   Create graph using Algorithm 5.1.1
2.   sort edges from the heaviest to the lightest
3.   for each edge $e_{i,j}$
4.     create vertex $\tau_k = \tau_i \cup \tau_j$
5.     for each vertex $\tau_m$
6.     if there are edges $e_{i,m}$ and $e_{j,m}$
7.       create edge $e_{k,m}$
8.       $w_{k,m} = \max(w_{i,m}, w_{j,m})$
9.     Remove $\tau_i$ and $\tau_j$
10.  end for
11.  return all $\tau$
```

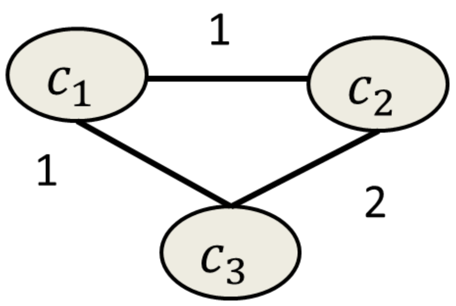
Example 5.1.2: Consider the SOP given by (5.1.3). The graph generation step is performed as previously described, resulting in the graph shown in Fig. 5.1. Furthermore, the merging of c_1 and c_3 into τ_5, given by (5.1.5), is also performed as previously described. The difference lies on the manner the edges for τ_5 are generated. Since neither c_1 nor c_3 has an edge to c_2, τ_5 does not have an edge to c_2. There is also no edge between τ_5 and c_4 because there is no edge connecting c_1 and c_4, even though there is an edge connecting c_3 and c_2. Therefore, the algorithm merges c_2 and c_4 into τ_6, given by (5.1.6), and the execution stops. The resulting SRBF is given by (5.1.7).

Example 5.1.2 illustrates a case when the simplified algorithm provides the same result as the original algorithm. However, in several cases, the solution quality is reduced. One reason is that the weight of edges may decrease as vertices are combined.

Example 5.1.3: This example illustrates a case when the simplified algorithm leads to a worse solution. Consider the following ISOP:

$$f = \overline{x_0}\,\overline{x_1}x_2x_3 + x_0\overline{x_2}x_4\overline{x_5} + x_1\overline{x_2}\,\overline{x_5}x_6 \tag{5.1.8}$$

Figure 5.2 – Graph for Example 5.1.3.



Source: The author.

In the following, we use:

$$c_1 = \overline{x_0}\,\overline{x_1}x_2x_3 \tag{5.1.9a}$$

$$\mathrm{c}_2 = x_0\overline{x_2}x_4\overline{x_5} \tag{5.1.9b}$$

$$c_3 = x_1\overline{x_2}\,\overline{x_5}x_6 \tag{5.1.9c}$$

The initial graph is shown in Fig. 5.2. Notice that the edges $e_{1,2}$ and $e_{1,3}$ exist because these combinations lead to one less reset operation. Since the graph is a clique, the version of the algorithm that does not update the edges weights returns a single RBF. The resulting RBF is as follows:

$$f = \overline{f_0}f_1 \tag{5.1.10a}$$

$$f_0 = \overline{x_0}\,\overline{x_1}\,\overline{x_2} + \overline{x_0}\,\overline{x_1}\,\overline{x_4} + \overline{x_0}\;\overline{x_2}\,\overline{x_6} + \overline{x_0}\;\overline{x_3}\,\overline{x_6} + \overline{x_1}\,\overline{x_2}\,\overline{x_4} + \overline{x_1}\,\overline{x_3}\,\overline{x_4} + \overline{x_2}\,\overline{x_4}\,\overline{x_6} + \overline{x_2}\,\overline{x_3}\,\overline{x_6} \tag{5.1.10b}$$

$$f_1 = \overline{x_0}\,\overline{x_1} + \overline{x_2}\,\overline{x_5} \tag{5.1.10c}$$

where

$$\overline{f_0} = \overline{(x_2x_3 + x_0x_4 + x_1x_6)} \tag{5.1.11}$$

There are 10 cubes in (5.1.10). Therefore, the evaluation of (5.1.10) takes 11 cycles, including the initial reset operation.

When the edges weights are updated, the behavior is different, as shown in the following. The first cubes to be merged are c_2 and c_3. The resulting term $\tau_4 = c_2 + c_3$ is the following:

$$\tau_4 = \overline{\tau_{4_0}}\tau_{4_1} \tag{5.1.12a}$$

$$\tau_{4_0} = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_6} + \overline{x_1}\,\overline{x_4} + \overline{x_4}\,\overline{x_6} \tag{5.1.12b}$$

$$\tau_{4_1} = \overline{x_2}\,\overline{x_5} \tag{5.1.12c}$$

Equation (5.1.12) takes six cycles to be evaluated. Therefore, if the target function is evaluated as $\tau_4 + c_1$, 10 cycles are used, being one less than the single RBF solution given by (5.1.10). In this sense, the weight of edge $e_{1,4}$ is -1. The clique based algorithm is unable to detect this variation on edge weight, leading to a worse result. In order to improve the quality of the clique based method, we add a verification step to ensure that the solution quality does not decrease whenever two groups are combined, as shown in Algorithm 5.1.4.

Algorithm 5.1.4 – Clique-based approach for SRBF synthesis which ensures monotonicity of solution quality.

```
     Input: SOP $f$
     Output: SRBF
1.   Create graph using Algorithm 5.1.1
2.   sort edges from the heaviest to the lightest
3.   for each edge $e_{i,j}$
4.     create vertex $\tau_k = \tau_i \cup \tau_j$
5.     if $|\tau_k| \leq 1 + |\tau_i| + |\tau_j|$
6.       for each vertex $\tau_m$
7.       if there are edges $e_{i,m}$ and $e_{j,m}$
8.         create edge $e_{k,m}$
9.         $w_{k,m} = \max(w_{i,m}, w_{j,m})$
10.      Remove $\tau_i$ and $\tau_j$
11.  end for
12.  return all $\tau$
```

5.2 SINGLE-CUBE FACTORED SUM-OF-RECURSIVE-BOOLEAN-FORMS

The methods discussed previously provide a sequence of instruction that can be evaluated using $n + 2$ RSD. In this section, we propose a form that leads to smaller sequence of instructions than those obtained from SRBF at the cost of extra RSDs. In particular, we consider single-cube factor SRBF (SC-FSRBF). A SC-FSRBF is given by:

$$f = \pi_1 f_1 + \cdots + \pi_m f_m \tag{5.2.1}$$

where each π_i is a product of positive literals and each f_i is an SRBF. The sequence of operations to compute a term $\pi_i f_i$ is similar to the evaluation of an SRBF with inclusions of a third work RSD which stores the complement of the π_i. An SC-FSRBF that requires $n + k$ RSD, where $k \geq 2$, is a k-SC-FSRBF.

Example 5.2.1: Consider an ISOP given by:

$$f = x_0 x_1 \overline{x_2} x_3 x_4 + x_0 x_1 \overline{x_5} x_6 + x_0 x_1 \overline{x_5} x_7 \qquad (5.2.2)$$

Equation (30) can be written in the form (29), with a single term, as follows:

$$f = x_0 x_1 \left(\overline{x_2} \overline{(\overline{x_3} + \overline{x_4})} + \overline{x_5} \overline{(\overline{x_6}\, \overline{x_7})} \right) \qquad (5.2.3)$$

The resulting sequence of operations is shown in Table 5.2.

Table 5.2 – Sequence of operations to evaluate (5.2.3).

Operation	y_1	y_2	y_3
$y_1 = 0, y_2 = 0, y_3 = 0$	0	0	0
$y_3 \mathrel{+}= \overline{x_0}$			$\overline{x_0}$
$y_3 \mathrel{+}= \overline{x_1}$			$\overline{x_0} + \overline{x_1}$
$y_2 \mathrel{+}= \overline{x_3}$		$\overline{x_3}$	
$y_2 \mathrel{+}= \overline{x_4}$		$\overline{x_3} + \overline{x_4}$	
$y_1 \mathrel{+}= \overline{x_2}\, \overline{y_2}\, \overline{y_3}$	$x_0 x_1 \overline{x_2} x_3 x_4$		
$y_2 = 0$		0	
$y_2 \mathrel{+}= \overline{x_6}\, \overline{x_7}$		$\overline{x_6}\, \overline{x_7}$	
$y_1 \mathrel{+}= \overline{x_5}\, \overline{y_2}\, \overline{y_3}$	$x_0 x_1 \overline{x_2} x_3 x_4 + x_0 x_1 \overline{x_5} x_6 + x_0 x_1 \overline{x_5} x_7$		

Source: The author.

In fact, (5.2.1) can be used to arbitrary number of $k \geq 2$ work RSDs. In such cases, each π_i is a product of positive literals and each f_i is either an SRBF or a $(k-1)$-SC-FSRBF, given by (5.2.1).

We extend the SOP-based algorithm for SRBF synthesis to consider SC-FSRBF. The extended algorithm is given in Algorithm 5.2.1. When cubes are combined, they can form either an SRBF or an SC-FSRBF. A group that represents an SRBF is a T-group. In turn, a group that represents an SC-FSRBF is a Φ-group. Finally, a group of a single cube is a σ-group. The type of a group τ is denoted by τ_t. Moreover, τ_{cp} denotes the set of all positive literals that are common to all cubes in τ. The edges have also different types. A T-edge creates a T-group and a Φ-edge creates a Φ-group. The type of an edge e is denoted by e_t.

During the graph generation stage, in addition to the evaluation of the RBF for each pair of cubes, we also check if there are positive literals in common to c_i and c_j. This step creates a Φ-edge. There is a Φ-edge connecting two groups τ_i and τ_j if there is at least one positive literal that appears on all cubes of $\tau_i \cup \tau_j$. Notice that there may be both a Φ-edge and a T-edge connecting two vertices. The graph creation algorithm is given in Algorithm 5.2.1.

Similarly to the synthesis of SRBF, we consider two approaches to synthesize SC-FSRBF. The first approach updates all edges for each grouping performed. The second approach combines the groups by evaluating whether the resulting group forms a clique. We use the edge type to determine whether two groups can be merged. In particular, we do not allow a T-group (*i.e.*, a SRBF) to be merged with any group through a Φ-edge. We observed that allowing a T-group to become a Φ-group negatively impacts the solution quality. However, transforming a Φ-group into a T-group does not reduces the solution quality.

Algorithm 5.2.1 – Graph generation for SC-FSRBF synthesis.

```
    Input: SOP $f$, extra number of devices $k$
    Output: Graph
1.  Create graph using Algorithm 5.1.1
2.  Mark all edges in the graph as T-edges
3.  for each pair of cubes $c_i$, $c_j$ in $f$
4.    if $k > 0$ and $cp_i \cap cp_j \neq \{\ \}$ then
5.      $\omega_{i,j} = |cp_i \cap cp_j|$
6.      create a Φ-edge $e_{i,j}$ with weight $\omega_{i,j}$
7.    end if
8.  end for
```

The update edges approach, shown in Algorithm 5.2.2, uses the same strategy to combine groups as performed in Algorithm 5.1.1. The main difference lies on the graph generation stage. Moreover, Algorithm 5.2.2 is recursively run for each Φ-group. In this second run, $k - 1$ extra devices are used.

Example 5.2.2: Consider an SOP given by the following expression:

$$f = x_0x_1x_2\overline{x_3} + x_0x_1x_2\overline{x_4} + x_0x_5x_6\overline{x_7} \tag{5.2.4}$$

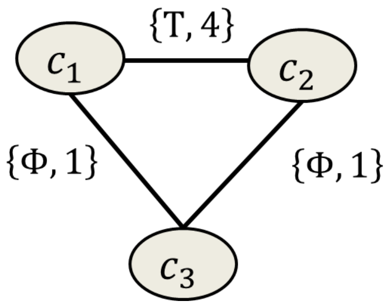
Let $k = 1$ be the number of extra RSD. The graph obtained through Algorithm 5.2.1 is shown in Fig. 5.3, where c_1, c_2 and c_3 are, respectively:

$$c_1 = x_0x_1x_2\overline{x_3} \tag{5.2.5a}$$

$$c_2 = x_0x_1x_2\overline{x_4} \tag{5.2.5b}$$

$$c_3 = x_0x_5x_6\overline{x_7} \tag{5.2.5c}$$

Figure 5.3 – Graph corresponding to Example 5.2.2.



Source: The author.

Algorithm 5.2.2 – SC-FSRBF synthesis using the update edges approach.

```
    Input: SOP $f$, extra number of devices $k$
    Output: FSRBF
1.  generate graph using algorithm 5.2.1
2.  sort edges from the heaviest to the lightest
3.  while there is an unvisited edge $e_{i,j}$
4.    if (${e_{i,j}}_t$ = Φ and (${\tau_i}_t$ = T or ${\tau_j}_t$ = T) )
5.      go to next edge
6.    $\tau_{i,j} = \tau_i \cup \tau_j$
7.    replace $\tau_i$ and $\tau_j$ by $\tau_{i,j}$
8.    create edges for $\tau_{i,j}$ (as in lines Algorithm 5.2.1)
9.    set type of $\tau_{i,j}$ accordingly to ${e_{i,j}}_t$
10. end while
11. if ($k > 0$)
12.   for each Φ-group $\tau = c_\tau \tau'$ ($c_\tau$ is the cube common to all cubes in $\tau$)
13.     call Algorithm 5.2.2 with $k' = k - 1$ for $\tau'$
14. return all $\tau$
```

The first pair of cubes to merged is c_1 and c_2. The resulting T-group is τ_4. Since literal x_0 appears in all cubes of τ_4 and c_3, there is a Φ-edge connecting τ_4 and c_3. Therefore, a Φ-group τ_5 is created comprising c_1, c_2 and c_3. Since $k > 0$ and τ_5 is a Φ-group, group τ_5' is obtained by factoring the common literal x_0 from τ_5. The resulting group $\tau_5{}'$ is the following:

$$\tau_5' = c_1' + c_2' + c_3' \tag{5.2.6a}$$

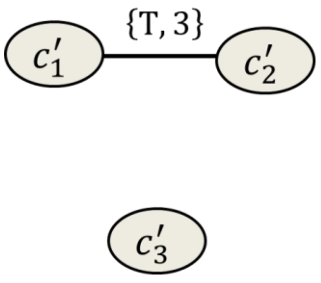
$$c_1' = x_1 x_2 \overline{x_3} \tag{5.2.6b}$$

$$c_2' = x_1 x_2 \overline{x_4} \tag{5.2.6c}$$

$$c_3' = x_5 x_6 \overline{x_7} \tag{5.2.6d}$$

The next step is to run Algorithm 5.2.2 for τ_5' with $k' = 0$. The resulting graph is shown in Fig. 5.4.

Figure 5.4 – Graph corresponding to (5.2.6).



Source: The author.

The only combination to be performed is merging c_1' and c_2' into τ_6. Hence, the final SC-FSRBF is as follows:

$$f = x_0\left((\overline{x_3} + \overline{x_4})\overline{(\overline{x_1} + \overline{x_2})} + \overline{x_7}\overline{(\overline{x_5} + \overline{x_6})}\right) \tag{5.2.7}$$

The resulting sequence of operations is shown in Table 5.3.

We also consider a clique-based algorithm for the synthesis of SC-FSRBF which is similar to Algorithm 5.1.3 and Algorithm 5.1.4. When an edge e with type e_t is selected, the algorithm evaluates whether the resulting group is a clique considering only the edges with same type e_t. Moreover, if e is a Φ-edge, then the clique is only valid if there is a positive cube c that is common to all vertices in the clique. In other words, whenever a Φ-edge is created there is a cube that is a factor of the corresponding expression.

Table 5.3 – Sequence of operations to evaluate the SC-FSRBF given by (5.2.7)

1.	$RESET\ (Y)$	
2.	$x_0 \rightarrow y_0$	$y_0 = \overline{x_0}$
3.	$x_1 \rightarrow y_1$	$y_1 = \overline{x_1}$
4.	$x_2 \rightarrow y_1$	$y_1 = \overline{x_1} + \overline{x_2}$
5.	$(y_0 + x_3 + y_1) \rightarrow y_2$	$y_2 = x_0x_1x_2\overline{x_3}$
6.	$(y_0 + x_4 + y_1) \rightarrow y_2$	$y_2 = x_0x_1x_2\overline{x_3} + x_0x_1x_2\overline{x_4}$
7.	$RESET\ y_1$	$y_1 = 0$
8.	$x_5 \rightarrow y_1$	$y_1 = \overline{x_5}$
9.	$x_6 \rightarrow y_1$	$y_1 = \overline{x_5} + \overline{x_6}$
10.	$(y_0 + x_7 + y_1) \rightarrow y_2$	$y_2 = x_0x_1x_2\overline{x_3} + x_0x_1x_2\overline{x_4} + x_0x_5x_6\overline{x_7}$

Source: The author.

Algorithm 5.2.3 – Synthesis of SC-FSRBF using the clique based approach.

```
    Input: SOP $f$, extra number of devices $k$
    Output: SC-FSRBF that can be evaluated with $n + k$ RSD.
1.  Create graph using Algorithm 5.2.1
2.  sort edges from the heaviest to the lightest
3.  for each edge $e_{i,j}$
4.    if ($e_{i,j}.t = \Phi$ and ($\tau_i.t =$ T or $\tau_2.t =$ T) )
5.      go to next edge
6.    if $e_{i,j}.t =$ T
7.      $\tau_k = \tau_i \cup \tau_j$
8       if $\tau_k$ is a clique, considering only T-edges, then
9.        merge $\tau_i$ and $\tau_j$ into a T-group $\tau_k$
10.     end if
11.   else
12.     if ${\tau_i}_{cp} \cap {\tau_j}_{cp} \neq \{\ \ \}$ then
13.       merge $\tau_i$ and $\tau_j$ into a $\Phi$-group $\tau_k$
14.     end if
15.   end if
16. end for
17. for each $\Phi$-group $\tau = c_\tau \tau'$ ($c_\tau$ is the cube common to all cubes in $\tau$)
18      call Algorithm 5.2.2 with $k' = k - 1$ for $\tau'$
19. return all $\tau$
```

Example 5.2.3: Consider the following ISOP:

$$f = x_0x_1x_2x_3\overline{x_4} + x_0x_1\overline{x_2}x_4x_5 + \overline{x_0}x_2x_4x_5x_6 + x_6x_7\overline{x_8} \quad (5.2.8)$$

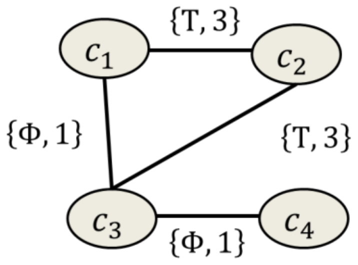
In the following:

$$c_1 = x_0x_1x_2x_3\overline{x_4} \quad (5.2.9a)$$

$$c_2 = x_1\overline{x_2}x_4x_5 \quad (5.2.9b)$$

$$c_3 = \overline{x_0}x_2x_4x_5x_6 \quad (5.2.9c)$$

Figure 5.5 – Graph for Example 5.2.3.



Source: The author.

The resulting graph is shown in Fig. 5.5. Let $e_{1,2}$ be the first visited edge, resulting in the merging of c_1 and c_2. The next edge is $e_{2,3}$. However, vertices c_2 and c_3 cannot be merged because the resulting group $\{c_1, c_2, c_3\}$ does not form a clique considering only T-edges. Notice that there is only a Φ-edge connecting c_1 and c_2. The next edge to be visited is $e_{1,3}$. However, c_1 and c_3 are not combined because no positive literal appears in all c_1, c_2 and c_3. Finally, the vertices c_3 and c_4 are merged through edge $e_{3,4}$ and the algorithm execution stops. The final expression is shown in the following and it can be evaluated in 18 cycles.

$$f = \tau_1 + \tau_2 \quad (5.2.10)$$

where τ_1 is an RBF and τ_2 is an SC-FSRBF, which are given, respectively, by:

$$\tau_1 = x_0x_1x_2x_3\overline{x_4} + x_0x_1\overline{x_2}x_4x_5 = (\overline{x_4} + \overline{x_2})\overline{(\overline{x_0} + \overline{x_1} + \overline{x_2}\,\overline{x_4} + \overline{x_2}\,\overline{x_5} + \overline{x_3}\,\overline{x_4} + \overline{x_3}\,\overline{x_5}\,)} \quad (5.2.11)$$

and

$$\tau_2 = \overline{x_0}x_2x_4x_5x_6 + x_6x_7\overline{x_8} = x_6(\overline{x_0}x_2x_4x_5 + x_7\overline{x_8}) \quad (5.2.12)$$

5.3 EXPERIMENTAL RESULTS

We begin by comparing the different variations of SRBF synthesis. The results for the 15 functions which require more cycles are summarized in Table 5.4. When the clique-based variations, described in Algorithm 5.13 and Algorithm 5.14, are compared to Algorithm 5.1.2, the average number of cycles increases 40% and 23%, respectively. In turn, the average runtime is reduced by 75% and 70%, respectively. Information on the benchmark is presented in Table 4.6.

Table 5.4 – Comparison among different algorithms proposed for SRBF synthesis.

	Algorithm 5.1.2		Algorithm 5.1.3		Algorithm 5.1.4	
	Cycles	Runtime (s)	Cycles	Runtime (s)	Cycles	Runtime (s)
pla (1)	1310	37.3	1782	9.07	1543	10.28
t481 (0)	994	7.58	1520	2.12	1355	2.17
intb (6)	542	1.65	750	0.48	687	0.48
alu4 (7)	477	0.72	689	0.27	570	0.28
test2 (24)	440	0.82	609	0.21	547	0.22
test2 (13)	428	0.76	586	0.19	525	0.21
apex2 (2)	419	9.91	701	2.39	558	5.47
test2 (8)	417	0.75	567	0.19	527	0.21
test2 (15)	413	0.67	546	0.17	475	0.18
test2 (0)	412	0.66	565	0.17	493	0.2
test2 (22)	405	0.57	499	0.16	471	0.17
test2 (18)	397	0.58	517	0.15	476	0.17
test2 (14)	396	0.64	507	0.17	477	0.17
test2 (19)	395	0.63	513	0.17	462	0.18
test2 (29)	395	0.64	560	0.16	473	0.18
AVERAGE	522	4.25	727	1.07	642	1.37

Source: The author.

The next comparison is between RBF and SRBF. In Table 5.5, we compare the DEC-RBF synthesis, discussed in Chapter 4, to the SRBF synthesis, given by Algorithm 5.1.2. The results are summarized in Table 5.5. The use of SRBF leads to huge improvements in terms of the number of cycles and execution time.

The justification for the runtime reduction is based on the size of RBF obtained. When the SOP is transformed into an RBF, each cube added to the RBF can make the size of the RBF to increase by a factor of n. Hence, the total number of cycles to evaluate m cubes as a single RBF is n^m. On the other hand, the initial cost for the SRBF takes into account each cube individually. Each cube needs at most $2 + n$ cycles to be evaluated. Hence, when m cubes are merged into a RBF, the resulting RBF requires at most $m(2 + n)$ cycles to be evaluated. Therefore, the size of each RBF within an SRBF is linear with respect to the number of cubes the RBF represents. This prevents the exponential growth of RBF and leads to the observed improvements regarding the synthesis time.

Table 5.5 – RBF and SRBF comparison for the 15 largest functions in the benchmark set.

Design (output)	RBF		SRBF	
	Cycles	Runtime (s)	Cycles	Runtime (s)
cordic (1)	35367	1717.68	1310	37.30
signet (1)	34601	3076.90	100	0.01
cordic (0)	29448	4650.32	126	0.22
signet (2)	14677	344.18	81	0.01
signet (0)	8484	186.40	90	0.01
t481 (0)	5067	86.05	994	7.58
apex2 (0)	3486	283.79	340	1.94
x6dn (1)	2508	21.79	91	0.04
vg2 (4)	2291	8.01	105	0.02
vg2 (5)	2255	7.43	105	0.02
vtx1 (1)	2206	7.34	105	0.02
x1dn (1)	2206	7.33	105	0.02
x9dn (2)	2206	7.32	105	0.02
apex1 (22)	1929	12.39	169	0.09
apex2 (2)	1753	147.21	419	9.91

Source: The author.

We also analyze simpler functions for which the number of cycles is limited to 150. The results are summarized in Table 5.6. Again, the use of SRBF leads to significant improvements in terms of the number of cycles while preserving the algorithm execution time.

Table 5.7 summarizes the results obtained for SC-FSRBF. The number of extra RSD *k* ranges from 0 to 4. Notice that, when *k*=0, the SC-FSRBF becomes an SRBF. We can observe that the improvements obtained by increasing *k* are mostly limited to *k=1*. We can also observe some anomalies where increasing *k* reduces the solution quality. For instance, the number of cycles to evaluate the output (6) of the design ‘intb’ increases from 542 to 555 when *k* goes from 0 to 1. Nevertheless, changing *k* from 1 to 2, improves the solution to 508 cycles. Another interesting case is the output (29) of the ‘test2’ design where the solution quality only decreases from the initial solution.

Table 5.6 – RBF and SRBF comparison for the 15 largest functions in the benchmark set limited to 150 cycles.

Design (output)	RBF		SRBF	
	cycles	Runtime (s)	cycles	Runtime (s)
table5 (10)	150	0.01	62	< 0.01
prom1 (15)	150	0.03	120	0.05
ti (4)	149	0.11	71	0.01
bcd (23)	147	0.02	85	< 0.01
prom1 (13)	146	0.03	120	0.05
apex1 (34)	144	0.08	88	0.01
intb (1)	144	0.07	99	0.02
prom (19)	144	0.03	107	0.05
alu4 (2)	143	0.07	99	0.02
bca (31)	143	0.01	76	< 0.01
bcb (16)	143	0.01	74	< 0.01
bcb (30)	143	0.01	76	< 0.01
max512 (5)	143	0.06	132	0.05
xparc (37)	142	0.04	92	0.06
prom1 (20)	139	0.03	124	0.05

Source: The author.

Table 5.7 – FC-SRBF solutions for the 15 most complex functions.

	k=0		*k*=1		*k*=2		*k*=3		*k*=4	
Design (out)	Cycles	Runtime (s)	Cycles	Runtime (s)	Cycles	Runtime (s)	Cycles	Runtime (s)	Cycles	Runtime (s)
cordic (1)	1310	37.57	1274	89.41	1236	93.32	1237	93.68	1237	94.03
t481 (0)	994	7.53	979	21.3	972	21.32	972	21.33	972	21.35
intb (6)	542	1.64	555	9.47	508	11.17	504	11.21	512	11.23
alu4 (7)	477	0.72	464	1.48	455	1.52	452	1.52	452	1.53
test2 (24)	440	0.83	441	1.27	437	1.29	434	1.28	434	1.29
test2 (13)	428	0.76	422	1.31	430	1.36	428	1.3	428	1.31
apex2 (2)	419	9.91	421	26.11	423	27.53	422	27.72	422	27.43
test2 (8)	417	0.75	404	1.33	406	1.34	406	1.34	406	1.33
test2 (15)	413	0.67	407	1.17	404	1.16	405	1.16	405	1.16
test2 (0)	412	0.66	419	1.19	416	1.14	416	1.13	416	1.14
test2 (22)	405	0.57	392	0.89	388	0.88	388	0.89	388	0.88
test2 (18)	397	0.58	393	0.98	392	0.99	392	0.99	392	0.99
test2 (14)	396	0.64	411	1.12	408	1.12	408	1.13	408	1.12
test2 (19)	395	0.64	399	0.97	395	0.99	395	0.98	395	0.99
test2 (29)	395	0.64	407	0.98	412	0.98	411	0.99	411	0.99

Source: The author.

6 OTHER CONTRIBUTIONS

In this chapter, we present two other contributions regarding the logic synthesis for RSD-IMP logic structure. In Section 6.1, we explore the benefits of having the variables available in an input RSD in both negative and complemented forms. In Section 6.2, we discuss the logic design of a full-adder (FA) and ripple-carry adder (RCA) in RSD-IMP logic.

6.1 COMPLEMENTED INPUTS APPROACH

In this section, we consider that each variable is available in an input RSD in both direct and complemented forms. Therefore, each cube in the SOP can be evaluated in a single cycle. For each variable x_i that is represented by at least one direct literal in the SOP, an operation of the form $x_i \rightarrow y_k$ is performed. Therefore, an SOP with m cubes can be evaluated in at most $1 + m + n$ cycles. In contrast to the methods previously discussed, does not depend directly on the number of positive literals. Therefore, we expect this approach to reduce the average length of the sequence of instructions. However, in some cases, SRBF and FSRBF can lead to better solutions.

Example 6.1.1: Consider the following SOP *f*:

$$f = x_0x_1 + x_1x_2 + x_0x_2 \qquad (6.1.1)$$

Using the complemented inputs approach, the sequence of instructions described in Table 6.1 is obtained, and seven instructions are required. Instructions 2 through 4 are used to complement the input variables while each one of the final three instructions evaluates a cube in (6.1.1). However, if (6.1.1) is written as a negative phase RBF, only five instructions are used, as shown in Table 6. 2.

Table 6.1 – Sequence of operations to evaluate (6.1.1) using the complemented inputs method.

$RESET(Y)$	
$x_0 \rightarrow y_0$	$y_0 = \overline{x_0}$
$x_1 \rightarrow y_1$	$y_1 = \overline{x_1}$
$x_2 \rightarrow y_2$	$y_2 = \overline{x_2}$
$(y_0 + y_1) \rightarrow y_3$	$y_3 = x_0x_1$
$(y_0 + y_2) \rightarrow y_3$	$y_3 = x_0x_1 + x_0x_2$
$(y_1 + y_2) \rightarrow y_3$	$y_3 = x_0x_1 + x_0x_2 + x_1x_2$

Source: The author.

Table 6.2 – Sequence of operations to evaluate (6.1.1) as an RBF.

$RESET(Y)$	
$(x_0 + x_1) \rightarrow y_0$	$y_0 = \overline{x_0}\,\overline{x_1}$
$(x_0 + x_2) \rightarrow y_0$	$y_0 = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2}$
$(x_1 + x_2) \rightarrow y_0$	$y_0 = \overline{x_0}\,\overline{x_1} + \overline{x_0}\,\overline{x_2} + \overline{x_1}\,\overline{x_2}$
$y_0 \rightarrow y_1$	$y_1 = x_0x_1 + x_0x_2 + x_1x_2$

Source: The author.

The number of cycles to evaluate an SOP f through the complemented inputs approach can be reduced through a factoring process, aiming to reduce the total number of cubes.

Example 6.1.2: Consider the following SOP *f*:

$$f = x_0x_1x_6 + x_0x_1x_7 + x_0x_1x_8 + x_2x_3x_6 + x_2x_3x_7 + x_2x_3x_8 + x_4x_5x_6 + x_4x_5x_7 + x_4x_5x_8 \tag{6.1.2}$$

Since (6.1.2) comprises nine cubes and nine variables represented by a positive literal, the resulting number of cycles to evaluate (6.1.2) is 19. The number of cycles can be reduced by factoring (6.1.2), as follows:

$$f = (x_0x_1 + x_2x_3 + x_4x_5)(x_6 + x_7 + x_8) \tag{6.1.3}$$

Equation (6.1.3) comprises six cubes. The resulting sequence of instructions to evaluate (6.1.3) is shown in Table 6.3. For sake of simplicity, we skip the initial 10 instructions that correspond to the reset operations and to the negation of inputs. Hence, the total number of instructions to evaluate (6.1.3) is 16. The main idea is to rewrite (6.1.3) as follows:

$$f = \overline{f_1}(x_6 + x_7 + x_8) \tag{6.1.4}$$

where

$$\overline{f_1} = \overline{(x_0x_1 + x_2x_3 + x_4x_5)} \tag{6.1.5}$$

The first three steps shown in Table 6.3 store $\mathrm{f_1}$ at $\mathrm{y_0}$, and then $\overline{\mathrm{f_1}}$ is stored at $\mathrm{y_1}$. Finally, for each cube in $\mathrm{x_6 + x_7 + x_8}$, one operation is performed to store $\mathrm{f_1x_6}$, $\mathrm{f_1x_7}$ and $\mathrm{f_1x_8}$ into $\mathrm{y_2}$.

Table 6.3 – Sequence of operations to evaluate (6.1.3).

$(\overline{x_0} + \overline{x_1}) \to y_0$	$y_0 = x_0x_1$
$(\overline{x_2} + \overline{x_3}) \to y_0$	$y_0 = x_0x_1 + x_2x_3$
$(\overline{x_4} + \overline{x_5}) \to y_0$	$y_0 = x_0x_1 + x_2x_3 + x_4x_5$
$y_0 \to y_1$	$y_1 = \overline{x_0x_1 + x_2x_3 + x_4x_5}$
$(\overline{x_6} + y_1) \to y_2$	$y_2 = (x_0x_1 + x_2x_3 + x_4x_5)x_6$
$(\overline{x_7} + y_1) \to y_2$	$y_2 = (x_0x_1 + x_2x_3 + x_4x_5)(x_6 + x_7)$
$(\overline{x_8} + y_1) \to y_2$	$y_2 = (x_0x_1 + x_2x_3 + x_4x_5)(x_6 + x_7 + x_8)$

Source: The author.

We apply a standard double-cube divisor technique and modify the method to evaluate the solution improvement. Instead of counting the reduction on the number of literals, we count the reduction on the number of cubes. As a consequence, we only consider divisions where both the divisor and the quotient have at least two cubes.

Example 6.1.3: To illustrate the difference between the standard literal oriented division and a cube oriented division, consider the following SOP f with seven cubes:

$$f = x_0x_1x_2x_3x_4 + x_0x_1x_2x_3x_5 + x_0x_1x_2x_3x_6 + x_4x_7 + x_5x_7 + x_4x_8 + x_5x_8 \tag{6.1.6}$$

An optimal factored form for (6.1.6), in terms of literal count, comprises literals, as follows:

$$f = x_0x_1x_2x_3(x_4 + x_5 + x_6) + (x_4 + x_5)(x_7 + x_8) \tag{6.1.7}$$

Notice that (6.1.7) comprises eight cubes, being one more than (6.1.6). Therefore, reducing the number of literals is not always a good strategy to minimize the number of cubes. A cube oriented division could yield the following expression:

$$f = (x_4 + x_5)(x_0x_1x_2x_3 + x_7 + x_8) + x_0x_1x_2x_3x_6 \tag{6.1.8}$$

Equation (6.1.8) comprises 13 literals, two more than (6.1.7), but six cubes. In this sense, a cube oriented division reduced the number of cubes by one when compared to the original SOP given by (6.1.6).

Algorithm 6.1.1 describes the division process. The algorithm selects the lines with most number of '1'. This line represents a rectangle. Then, the algorithm searches for the line that maximizes the gain when added to a rectangle and adds this line to the rectangle. When there are no lines to be added, the rectangle is a prime rectangle that represents a divisor and a quotient for *f*. The process repeats until no good divisors are found.

Algorithm 6.1.1 – Division algorithm.

```
1.  Create matrix from the cube intersections
2.  L1=all lines in the table
3.  L2=all lines in the table
4.  C={}
5.  while (L1 is not empty and C does not contain all cubes in f )
6.    select l in L1 with most number of columns in '1' that corresponds to cubes not in C
7.    start a rectangle R with l
8.    while there is a l2 in L2 that improves the solution
9.      add the line lmax in L2 that maximizes the gain to R
10.   end while
11.   if R contains more than one line and one column
12.     perform the division
13.     add all cubes impacted by the division to C
14.     remove from L1 all lines that do not have a '1' in at least one position
    corresponding to a cube not in C
15.   Else
16.     remove l from L1
17.   end if
18. end while
19. Run the algorithm recursively for each divisor and quotient found
20. Return
```

Table 6.4 presents the comparison among SC-FSRBF, the complemented inputs method (COMP), the division method (DIV) and the method described in (XIE, 2017). The COMP column presents the results obtained without the division process, and the DIV column presents the results after the division process. The solutions from the SC-FSRBF are those that lead to the smallest number of cycles. As can be observed, even though the number of RSDs used by the complemented inputs approach is much higher compared to the SC-

FSRBF, the reduction on the number of cycles is also significant. Clearly, Algorithm 6.1.1 has a huge drawback related to the execution time. Compared to the method proposed in (XIE, 2017), we obtain a good trade-off between the number of cycles and the number of RSDs. See Table 4.6 for more information on the benchmark set.

Table 6.4 – Comparison of the complemented inputs approach to SC-FCSRBF.

Design (out)	SC-FSRBF			COMP		DIV			(XIE. 2017)	
	Cycles	RSD	Runtime (s)	Cycles	RSD	Cycles	RSD	Runtime (s)	Cycles	RSD
cordic (1)	1236	26	93.32	792	44	62	66	8315	7	36284
apex2 (2)	419	41	9.91	548	64	166	121	2364	7	37204
t481 (0)	972	20	21.32	498	33	100	77	565.7	7	15906
apex2 (0)	331	43	3.53	296	57	161	118	566.0	7	20367
apex2 (1)	270	43	4.6	279	54	133	106	667.7	7	19345
alu4 (7)	452	19	1.52	197	29	165	54	9497	7	5307
alu4 (4)	332	17	1.25	196	29	174	63	257.8	7	5278
cordic (0)	122	26	1	167	47	111	63	0.72	7	6768
misex3 (7)	274	19	1.6	156	29	116	51	94.63	7	4118
misex3 (3)	263	17	0.83	147	29	109	48	117.5	7	3857
rd84 (1)	199	10	0.35	137	17	59	36	4.82	7	2193
misex3 (2)	239	16	0.29	135	29	117	60	52.17	7	3509
max1024 (5)	252	13	0.31	132	21	132	22	494.9	7	2562
misex3 (13)	219	16	0.27	131	29	128	36	27.7	7	3393
seq (5)	155	46	0.89	129	65	116	71	18.95	7	8162

Source: The author.

6.2 LOGIC DESIGN OF BINARY ADDER

In this section, we discuss the logic design of a full-adder (FA) and a ripple-carry adder (RCA) in RSD-IMP logic structure. Initially, we focus on the standard IMP gate, as previously detailed. Then, we discuss improvements to the RCA design by exploring the memory matrix. Finally, we evaluate the possibility to perform several independent sums simultaneously in the memory matrix.

Hereafter, we consider the task of adding two n-bit positive integers A and B. We use a_i and b_i to refer to the ith bit of A and B, respectively.

Our first contribution is a novel scheme for a FA design which fully exploits multi-input implication. This FA implementation is used as basis for the variations of RCA proposed herein.

6.2.1 Full-adder design

In the following, devices a, b and c are the three inputs for the FA. The sum and the carry-out outputs are stored at sum and $cout$, respectively. We also use an auxiliary device tmp. The FA implementation is based on writing the XOR3 as follows:

$$a \oplus b \oplus c = abc + \overline{(ab + ac + bc)}(a + b + c) \qquad 6.2.1$$

Since the term $ab + ac + bc$ corresponds to the majority function that represents the carry-out output, a key idea in FA design is to exploit the logic sharing between the sum and carry outputs. Since both the XOR3 and the majority functions are self-dual functions, we can rewrite (6.2.1) as follows:

$$a \oplus b \oplus c = \overline{\bar{a}\,\bar{b}\,\bar{c} + \overline{\left(\bar{a}\,\bar{b} + \bar{a}\,\bar{c} + \bar{b}\,\bar{c}\right)}(\bar{a} + \bar{b} + \bar{c})} \qquad 6.2.2$$

Equation (6.2.2) can be computed as shown in Table 6.5. The first step is to reset the state of devices *sum*, $cout$ and tmp. Then, the term $\bar{a}\,\bar{b} + \bar{a}\,\bar{c} + \bar{b}\,\bar{c}$, that is the complement of the carry out output, is stored into tmp. The following steps perform the AND operation between $\left(\bar{a}\,\bar{b} + \bar{a}\,\bar{c} + \bar{b}\,\bar{c}\right)$ and $(\bar{a} + \bar{b} + \bar{c})$, and store the result into *cout*. Then, cube $\bar{a}\,\bar{b}\,\bar{c}$ is added to $cout$. Then, the complement of *cout* is stored into the *sum* device which holds the final value for the sum output. Finally, the value of *tmp* is complemented and stored into *cout* such that *cout* contains the final value for the carry-out.

Table 6.5 – Cycles to evaluate a FA based on (6.2.2)

1.	$sum = 0, cout = 0, tmp = 0$	
2.	$(a + b) \rightarrow tmp$	$tmp = \bar{a}\bar{b}$
3.	$(a + c) \rightarrow tmp$	$tmp = \bar{a}\bar{b} + \bar{a}\bar{c}$
4.	$(b + c) \rightarrow tmp$	$tmp = \bar{a}\bar{b} + \bar{a}\bar{c} + \bar{b}\bar{c}$
5.	$(a + tmp) \rightarrow cout$	$cout = \bar{a}bc$
6.	$(b + tmp) \rightarrow cout$	$cout = \bar{a}bc + a\bar{b}c$
7.	$(c + tmp) \rightarrow cout$	$cout = \bar{a}bc + a\bar{b}c + ab\bar{c}$
8.	$(a + b + c) \rightarrow cout$	$cout = \bar{a}bc + a\bar{b}c + ab\bar{c} + \bar{a}\bar{b}\bar{c}$
9.	$cout \rightarrow sum$	$sum = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$
10.	$RESET(cout)$	$cout = 0$
11.	$tmp \rightarrow cout$	$cout = ab + ac + bc$

Source: The author.

The total number of cycles to compute both outputs of the FA is 11, and six devices are used. The benefits obtained from using more devices appear to be limited. In particular, an extra device can be used to skip the reset operation at line 10 in Table 6.2.

6.2.2 Single-line RCA

In this section, we consider two RCA implementations where all RSDs are placed into a single memory line. The first implementation obtains the minimum number of devices whereas the second implementation improves the delay by adding more devices. In the following, A and B are the two numbers to be summed, with *n* bits each. The *i*-th bit of A and B are represented by a_i and b_i, respectively.

Considering that the first stage of the RCA is a FA, there are *2n+1* input devices to store A, B and the carry-in for the first stage. The sum bits and the carry-out signal of the last stage add $n + 1$ devices. Therefore, there are $3n + 2$ devices, representing all primary inputs and primary outputs, used in all implementations.

In order to minimize the number of RSDs, we use the same device to store all carry signals. Moreover, two auxiliary devices ($tmp0$ and $tmp1$) are used for all bits of the RCA. Therefore, this implementation uses $3n + 4$ devices. The computation scheme is the same as for the FA described in Table 6.5. The cycles are shown in Table 6.6. The number of operations is $11n$.

Table 6.6 – Cycles to evaluate a FA in the single line RCA.

1.	$tmp0 = 0, tmp1 = 0,$ $sum_i = 0$	
2.	$(a_i + b_i) \rightarrow tmp0$	$tmp0 = \bar{a}_i\bar{b}_i$
3.	$(a_i + carry) \rightarrow tmp0$	$tmp0 = \bar{a}_i\bar{b}_i + \bar{a}_i\bar{c}_i$
4.	$(b_i + carry) \rightarrow tmp0$	$tmp0 = \bar{a}_i\bar{b}_i + \bar{a}_i\bar{c}_i + \bar{b}_i\bar{c}_i$
5.	$(a_i + tmp0) \rightarrow tmp1$	$tmp1 = \bar{a}_i b_i c_i$
6.	$(b_i + tmp0) \rightarrow tmp1$	$tmp1 = \bar{a}_i b_i c_i + a_i\bar{b}_i c_i$
7.	$(c + tmp0) \rightarrow tmp1$	$tmp1 = \bar{a}_i b_i c_i + a_i\bar{b}_i c_i + a_i b_i\bar{c}_i$
8.	$(a_i + b_i + carry) \rightarrow tmp1$	$tmp1 = \bar{a}_i b_i c_i + a_i\bar{b}_i c_i + a_i b_i\bar{c}_i + \bar{a}_i\bar{b}_i\bar{c}_i$
9.	$tmp1 \rightarrow sum_i$	$sum_i = a_i\bar{b}_i\bar{c}_i + \bar{a}_i b_i\bar{c}_i + \bar{a}_i\bar{b}_i c_i + a_i b_i c_i$
10.	$carry = 0$	$carry = 0$
11.	$tmp0 \rightarrow carry$	$carry = a_i b_i + a_i c_i + b_i c_i$

Source: The author.

The delay can be improved by considering that each FA contains an independent auxiliary device. In this case, the reset operations are not required at each stage since a single reset can be performed for all devices at the beginning of the computation. If each FA contains a single auxiliary device, the number of operations is reduced to $10n + 1$ while the number of devices is $5n$. In turn, if each FA contains two auxiliary devices, the number of cycles can be further reduced to $9n + 1$ while the number of devices increases to $6n$.

From the logic design perspective, expanding the single line RCA to perform several independent sums is straightforward. Each pair of sums is placed at a line and all the sums can occur in parallel. However, there are some electrical challenges related to sneak paths that have to be addressed.

6.2.3 Multiple-lines RCA

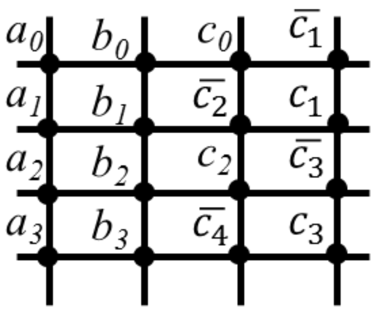
In this section, we aim to reduce the latency of the RCA by exploiting both dimensions of the crossbar structure. In contrast to the previous RCA implementation, we consider that each line corresponds to a FA such that a_i and b_i are stored in line i, as shown in Fig. 6.1. By placing each FA into a different line, some operations may occur in parallel. Despite such a parallelism, the adder still represents an RCA architecture because the carry signal propagates sequentially through the whole FA chain. The main steps of the multiple-lines RCA are as follows:

1) to compute the carry values sequentially;
2) to compute all sums of odd index in parallel; and
3) to compute all sums of even index in parallel.

6.2.3.1 Carry computation

Each carry signal is computed as described previously. As explained in the following, we use devices d_i and c_i to store the carry signals. If i is even, then c_i is the carry-in whereas for odd i, d_i is the carry-in, as illustrate in Fig 6.1. The sequence of operations to compute the carry for an even index is shown in Table 6.7. Notice that the operations performed in line 4 propagates the carry to the next line of the matrix.

Figure 6.1 – Carry-in configuration for the multiple-lines RCA.



Source: The author.

Table 6.7 – Carry computation for even index.

1.	$(a_i + b_i) \to d_i$
2.	$(a_i + c_i) \to d_i$
3.	$(a_i + c_i) \to d_i$
4.	$d_i \to d_{i+1}$

Source: The author.

Since the carry-in for odd *i* is stored at d_i, the carry-out computation for odd indexes is slightly different, as shown in Table 6.8.

Table 6.8 Carry computation for even index.

1.	$(a_i + b_i) \to c_i$
2.	$(a_i + d_i) \to c_i$
3.	$(a_i + d_i) \to c_i$
4.	$c_{i+1} \to c_{i+2}$

Source: The author.

The pattern of alternating *c* and *d* columns to store the carry signals continues throughout the chain. As a consequence, the carry signals are aligned such that all carry-in are stored in c_i for all even indexes *i* and in d_j for all odd indexes *j*. Since the first operation to compute the carry depends only on a_i and b_i, this operation can be performed in parallel for all indexes. Hence, the number of operations to compute all carry signals is $3n + 1$.

6.2.3.2 Sum output evaluation

After all carry values are computed, the sums are evaluated. It would be desirable to perform all such operations in parallel. However, the column of the carry value depends on whether the FA index is odd or even. Therefore, we begin by considering that all even bits are processed in parallel followed by the evaluation of all odd index. This procedure uses 10 cycles (five cycles for the even indexes and other five cycles for the odd indexes). The cycles for the even indexes and odd indexes are shown in Table 6.9.

Table 6.9 – Sum evaluation for even and odd indexes.

	Even indexes	Odd indexes
1.	$(a_i + d_i) \rightarrow e_i$	$(a_i + c_i) \rightarrow e_i$
2.	$(b_i + d_i) \rightarrow e_i$	$(b_i + c_i) \rightarrow e_i$
3.	$(c_i + d_i) \rightarrow e_i$	$(d_i + c_i) \rightarrow e_i$
4.	$(a_i + b_i + c_i) \rightarrow e_i$	$(a_i + b_i + d_i) \rightarrow e_i$
5,	$e_i \rightarrow f_i$	$e_i \rightarrow f_i$

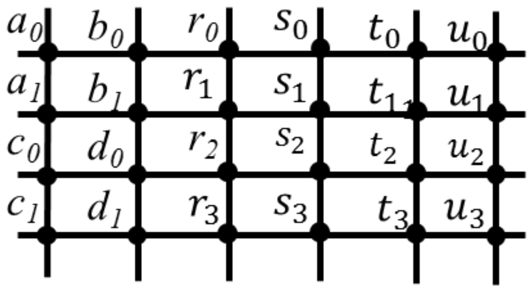
Source: The author.

The number of devices used is $6n$. In turn, the total number of cycles to perform the sum of A and B comprises the initial reset, $3n + 1$ cycles to evaluate all carry-out signals, five cycles to evaluate the sum of even indexes and more five cycles to evaluate the sum of odd indexes. Therefore, the number of cycles for $n \geq 2$ is:

$$12 + 3n \tag{6.2.3}$$

We consider two approaches to expand this idea to m independent sums in parallel. The first approach considers that all operands are aligned as shown in Fig. 6.2, which represents the structure used to perform the sums A+B and C+D, where each operand has two bits. Devices r_i, s_i, t_i and u_i are the auxiliary devices. In particular, r_0 and r_2 store the carry-in for the first FA of each sum.

Figure 6.2 – Crossbar structure to implement sums A+B and C+D in parallel using the multiple-lines RCA.


a_0 b_0 r_0 s_0 t_0 u_0
a_1 b_1 r_1 s_1 t_1 u_1
c_0 d_0 r_2 s_2 t_2 u_2
c_1 d_1 r_3 s_3 t_3 u_3

Source: The author.

The structure shown in Fig. 6.2 allows most of the operations needed to compute A+B to be performed in parallel to the operations for C+D. However, the last cycle of the carry evaluation (line 4 in Table 6.7 and in Table 6.8) cannot be performed in parallel for both sums. Considering the case shown in Fig. 6.2, it is not possible to transfer s_0 to s_1 at the cycle that s_2 is sent to s_3. Hence, this step is performed sequentially for all sums. The carry computation phase uses $2n+1$ cycles that correspond to the first three cycles in Table 6.7 and in Table 6.8, plus mn cycles to propagate the carry signals for the next line. The sum evaluation phase requires more 10 cycles, as in the single-line RCA version. Therefore, the total number of cycles to perform m sums, where each operand is n-bits wide is as follows:

$$12 + 2n + mn \qquad (6.2.4)$$

Notice that, the number of cycles increases with the number of operands. In contrast, the number of cycles for the single-line implementation is independent from m. Therefore, for a fixed number of bits n, there is a maximum number of sums m (m_{max}) such that the multiple-line implementation is more efficient than the single-line one. For $n \geq 2$, m_{max} is given by:

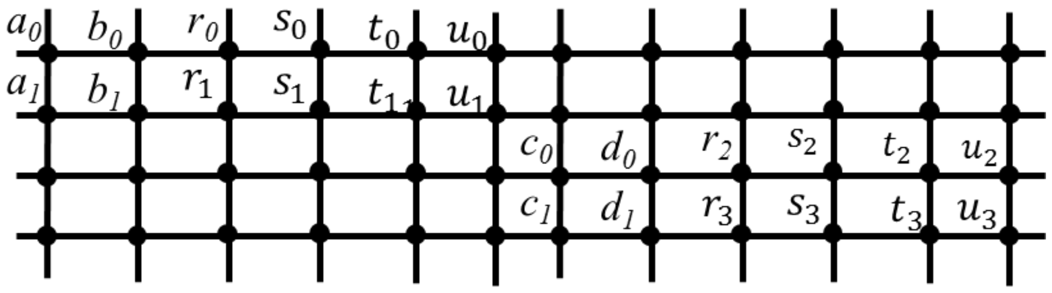
$$m_{max} = \frac{7n - 11}{n} \qquad (6.2.4)$$

As n increases, m_{max} approaches 7:

$$\lim_{n \to \infty} m_{max} = 7 \qquad (6.2.5)$$

The second variation aims to overcome the limitation of the previous strategy by placing the sums in a diagonal, as illustrated in Fig. 6.3. In this approach, the sums are independent from each other. Hence, the number of cycles is independent from m, being given by (5.3.1). In turn, the number of devices increases from $6nm$ to $6nm^2$.

Figure 6.3 – Alternative crossbar structure to implement sums A+B and C+D in parallel using the multiple-lines RCA.



Source: The author.

The RCA design is compared to different implementations of RCA based on RSD-NVM. The results are summarized in Table 6.10. We only consider the case of a single sum since this is the data presented in related works. Moreover, we also exclude implementations that modify the crossbar structure by adding transistors (SIEMON, 2015). Some works only discuss FA design. For these cases, we assume that the RCA consists of *n* copies of FA without considering optimizations that may exist. Overall, the proposed implementations obtain a good trade-off between the number of cycles and the number of RSDs. Moreover, the adder described in (YANG, 2016) also considers AND operations, as given by (3.2.24).

Table 6.10 – Comparison of the proposed adders to previous works.

	Style	Number of cycles	Number of RSD
Single line RCA	IMP	11*n*	3*n*+4
Multiple line RCA	IMP	12+3*n*	6*n*
(TALATI,2016)	Hybrid	*5n+18*	9*n*
(YANG, 2016)	IMP	14*n*	8*n*
(ALAMGIR, 2016)	Switch	20*n*	17*n*
(XIE, 2017)	Hybrid	7*n*/2	216*n*

Source: The author.

7 CONCLUSIONS AND FUTURE WORK

Advances of nanotechnologies bring new logic paradigms which do not always correspond to the standard switch based logic of standard MOSFET devices. In this sense, novel logic algorithms are being developed for a better understanding of these new paradigms. One among these paradigms is based on evaluating a given Boolean function as a sequence of basic instructions. Two of such instruction based approaches rely on majority logic and material implication logic.

This thesis focuses on developing algorithms for RSD-IMP logic structure. The overall approach consists on defining classes of Boolean expressions that have a direct translation to a sequence of instructions. In this way, it is possible to analyze how the number of instructions and the number of RSD depend on the size of such expressions. Consequently, logic synthesis algorithms can be tailored to optimized said expressions while taking into account the characteristics of RSD-IMP logic structure.

The first class of expressions considered, proposed in (LEHTONEN, 2010), is the class of RBF. The interesting property of this class is that such expressions can be translated into a sequence of operations that require only n+2 RSD to be evaluated. Our main contribution regarding RBF is a novel decomposition based method that, given a decomposition of h into f and g as $h = f \circ g$, where $\circ$ can be and AND or an XOR operator, combines the RBF for f and g to obtain RBF h.

We have expanded the RBF class to SRBF. SRBF can also be evaluated using the minimum number of $n+2$ RSDs and greatly reduces the number of instructions required to evaluate diverse Boolean functions. Then, we have further expanded SRBF to SC-FSRBF. In contrast to SRBF, SC-FSRBF requires more devices to be evaluated but can lead to smaller sequences of operations.

We continue our contributions by considering a different approach which is based on having all variables in both positive and negative polarities in the input RSD. Then, each cube in a given SOP can be evaluated in a single cycle. We also apply a division process over the input SOP to further reduce the length of instructions. Our last contributions regard the logic design of adders in RSD-IMP logic structure.

7.1 FUTURE WORK

About future work, it is worth to make some remarks related to logic synthesis for RSD-IMP logic structure.

In Section 5.2, we proposed the use of SC-FSRBF. This class of expressions is a subcase of a more general class, which is the factored SRBF (FSRBF). An FSRBF can be recursively defined as an RBF, a sum or a product of FSRBF. Notice that this definition is similar to a factored form where the literals are RBFs. The use of FSBRF should greatly reduce the number of cycles.

The algorithms can be expanded to handle multiple output functions. If the input is a multiple output SOP, an extraction algorithm, similar to the division procedure show in Algorithm 6.1.1 can be used to transform the SOP into a multilevel network. Then, each node in the network can be synthesized using any of the methods described herein.

Another option to take into account multiple output functions is to perform a LUT-based technology mapping over an AIG. In this case, for each cut enumerated, a sequence of instructions is generated using any of the methods described herein. This sequence of instructions is used to evaluate the cost of the cut in terms of the number of RSDs and the number of cycles. During the covering phase, the technology mapping chooses the best covering for the target AIG. We notice that the covering phase should differ from the applied one in standard LUT-based designs due to the following reasons:

1) The methods used to estimate the delay in conventional logic styles cannot be used to estimate the delay on the RSD-IMP logic structure. The reason for this is that in conventional combinational logic, the delay is determined by the worst case analysis whereas in RSD the delay is the sum of the delay of all cuts in the cover.
2) The area in conventional logic style is approximate by the number of LUTs in the design. In RSD-IMP logic structure, since RSD can be shared by many different LUTs, the area evaluation can be performed using the worst case analysis.

In this sense, the methods to estimate delay and area are somewhat reversed when comparing the conventional logic to the RSD-IMP logic structure.

The methods described herein can also be combined with f other works. For instance, in (XIE, 2017), each minterm of the target function is placed in a line. Then, all minterms are evaluated in parallel and summed. One possible approach to combine these methods is to obtain an SC-FSRBF. Then, the terms of the SC-FSRBF are placed into different lines such that the evaluation can be performed in parallel. This combination can provide a different trade-off in terms of the number of cycles and the number of RSDs.

REFERENCES

ADAM, G. C.; HOSKINS, B. D.; PREZIOSO, M.; STRUKOV, D.; B. Optimized stateful material implication logic for three-dimensional data manipulation. **NanoResearch**, v. 9, n. 12, p. 3914-3923, 2016.

AKERS, S.B. Binary Decision Diagrams. **IEEE Transactions on Computers**, v. C-27, n. 6, p. 509-516, 1978.

AKINAGA, H.; SHIMA, H. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. **Proceedings of the IEEE**, v. 98, n. 12, p. 2237-2251, Dec. 2010.

ALAMGIR, Z.; BECKMANN, K.; CADY, N.; VELASQUEZ, A. JHA, S. K. Flow-based computing on nanoscale crossbars: Design and implementation of full adders. **IEEE International Symposium on Circuits and Systems (ISCAS)**, p. 1870-1873, 2016.

AMIRSOLEIMANI, A.; AHMADI, M.; AHMADI, A. Logic Design on Mirrored Memristive Crossbars. **IEEE Transactions on Circuits and Systems II: Express Briefs**, 2017.

BORGHETTI, J.; SNIDER, G.S.; KUEKES, P.J.; YANG, J.J.; STEWART, D.R.; WILLIAMS, R.S. Memristive'switches enable 'stateful'logic operations via material implication. **Nature**, v. 464, n. 7290, p. 873-876, 2010.

BRAYTON, R.; McMULLEN, C. The decomposition and factorization of Boolean expressions. **in Proc. Of 2016 IEEE International Symposium on Circuits and Systems (ISCAS)**, pp. 29-54, 1982.

BRAYTON, R. K.; SANGIOVANNI-VINCENTELLI, A. L.; McMULLEN, C. T.; HACHTEL, G. D. **Logic Minimization Algorithms for VLSI Synthesis**, Kluwer Academic Publishers, Norwell, MA, 1984.

BRYANT, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, v. C-35, n. 8, p. 677-691, Aug. 1986.

CHAKRABORTI, S.; CHOWDHARY, P. V.; DATTA, K.; SENGUPTA, I. BDD based synthesis of Boolean functions using memristors. **2014 9th International Design and Test Symposium (IDT)**, Algiers, 2014, pp. 136-141.

CHATTOPADHYAY, A.; RAKOSI, Z. Combinational logic synthesis for material implication. **2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip**, Hong Kong, 2011, pp. 200-203.

CHUA, L. O. Resistance switching memories are memristors. **Applied Physics A**, v. 102, n. 4, p. 765-783, 2011.

DE MICHELI, G. **Synthesis and Optimization of Digital Circuits.** McGraw-Hill Science/Engineering, Math, 1994.

EDWARDS, A.H.; BARNABY, H.J.; CAMPBELL, K.A.; KOZICKI, M.N.; LIU, W., MARINELLA, M.J. Reconfigurable Memristive Device Technologies. **Proceedings of the IEEE**, v. 103, n. 7, p. 1004-1033, July 2015.

FAN, D.; SHARAD, M.; ROY, K. Design and synthesis of ultralow energy spin-memristor threshold logic. **IEEE Trans. on Nanotechnology**, v. 13, n. 3, pp. 574-583, 2014.

GAILLARDON P. E. The Programmable Logic-in-Memory (PLiM) computer. **2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)**, Dresden, 2016, pp. 427-432.

GAO, L.; ALIBART, F.; STRUKOV D. Programmable CMOS/memristor threshold logic. **IEEE Trans. on Nanotechnology**, v. 12, n. 2, pp. 115-119, 2013.

GIBBSONS, J. F.; BEADLE, W. E. Switching properties of thin NIO films. **Solid-State Electron.**, v. 7, n. 11, p.785 -790, 1964.

GUO, Y.; WANG, X.; ZENG, Z. A Compact memristor-CMOS hybrid Look-up-table Design and Potential Application in FPGA. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. PP, no. 99, pp. 1-1, 2017.

HICKMOTT, T.W. Low-Frequency Negative Resistance in Thin Anodic Oxide Films. **Journal of Applied Physics**, v. 33, n. 9, p. 2669-2682, 1962.

INDIVERI, G.; LIU, S. C. Memory and Information Processing in Neuromorphic Systems. **Proceedings of the IEEE**, v. 103, n. 8, p. 1379-1397, Aug. 2015.

JAMES, A. P.; FRANCIS, L. R. V. J.; KUMAR, D. S. Resistive threshold logic. **IEEE Trans. on Very Large Scale Integration (VLSI) Systems**, v. 22, n. 1, p. 190-195, Jan. 2014.

KAGARIS, D. MOTO-X: A Multiple-Output Transistor-Level Synthesis CAD Tool. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 35, n. 1, p. 114-127, 2016.

KENT, A.D.; WORLEDGE, D.C. A new spin on magnetic memories. **Nature Nanotechnology**, v. 10, p. 187–191, March 2015.

KIM, K.; SHIN, S.; KANG, S.-M. S. Field programmable stateful logic array. **IEEE Trans. on Computer-Aided Design on Integrated Circuits and Systems**, v. 30, n. 12, p. 1800-1813, 2011.

KULKARNI, N.; YANG, J.; SEO, J. S.; VRUDHULA, S. Reducing Power, Leakage, and Area of Standard-Cell ASICs Using Threshold Logic Flip-Flops. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 24, n. 9, pp. 2873-2886, 2016.

KVATINSKY, S.; WALD, N.; SATAT, G.; KOLDONY, A.; WEISER, U. C.; FRIEDMAN, E. G. MRL — Memristor Ratioed Logic. **13th International Workshop on Cellular Nanoscale Networks and their Applications**, 2012, pp. 1-6.

KVATINSKY, S.; BELOUSOV, D.; LIMAN, S.; SATAT, G.; WALD, N.; FRIEDMAN, E. G.; KOLDONY, A.; WEISER, U. C. MAGIC — memristor-aided logic. **IEEE Trans. on Circuits and Systems II: Express Briefs**, v. 61, n. 11, p. 895-899, Nov. 2014.

KVATINSKY, S.; SATAT, G.; WALD, N.; FRIEDMAN, E. G.; KOLDONY, A.; WEISER, U. C. Memristor-based material implication (IMPLY) logic: design principles and methodologies. **IEEE Trans. on Very Large Scale Integration (VLSI) Systems**, v. 22, n. 10, p. 2054-2066, Oct. 2014.

LEE, C.Y. Representation of Switching Circuits by Binary-Decision Programs. **Bell System Technical Journal**, v. 38, p. 985-999, 1959.

LEHTONEN, E.; POIKONEN, J. H.; LAIHO, M. Two memristors suffice to compute all Boolean functions. **Electronics Letters**, v. 46, n. 3, p. 239-240, Feb. 2010.

LEVY, Y.; BRUCK, J.; CASSUTO, Y.; FRIEDMAN, E.G.; KOLDONY, A.; YAAKOBI, E.; KVATINSKY, S. Logic operations in memory using a memristive Akers array. **Microelectronics Journal**, v. 45, n. 11, pp. 1429–1437, November 2014.

LINN, E.; ROSEZIN, R.; TAPPERTZHOFEN, S.; BÖTTGER, U.; WASER, R. Beyond von Neumann-logic operations in passive crossbararrays alongside memory operations. **Nanotechnology**, v. 23, n. 30, 2012.

MAHMOUDI, H.; WINDBACHER, T.; SVERDLOV, V.; SELBERHERR S. Implication logic gates using spin-transfer-torque-operated magnetic tunnel junctions for intrinsic logic-in-memory. **Solid-State Electronics**, v. 84, p. 191-197, June 2013.

MARRANGHELLO, F. S.; CALLEGARO, V.; MARTINS, M. G. A.; REIS, A. I.; RIBAS, R. P. Factored forms for memristive material implication stateful logic. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, v. 5, n. 2, p.267-278, June 2015.

MARRANGHELLO, F. S.; CALLEGARO, V.; MARTINS, M.G.A.; REIS, A.I.; RIBAS, R.P. SOP Based Logic Synthesis for Memristive IMPLY Stateful Logic. **Int'l Conf. on Computer Design (ICCD)**, New York, 2015.

MARTINS, M.G.A.; CALLEGARO, V.; MARRANGHELLO, F.S.; RIBAS R.P.; REIS, A.I. Majority-based logic synthesis for nanometric technologies. **IEEE 14th International Conference on Nanotechnology (IEEE-NANO)**, pp.256-261, 18-21 Aug. 2014.

MARTINS, M. G. A.; CALLEGARO, V.; MARRANGHELLO, F. S.; RIBAS, R. P.; REIS, A. I. Majority-based logic synthesis for nanometric technologies. **14th IEEE International Conference on Nanotechnology**, Toronto, ON, 2014, pp. 256-261.

MARTINS, M.; MARRANGHELLO, F.; FRIEDMAN, J.; SAHAKIAN, A.; RIBAS, R.P.; REIS, A. Enhanced Spin-Diode Synthesis Using Logic Sharing. **2015 Euromicro Conference on Digital System Design**, Funchal, 2015, pp. 218-224.

MARTINS, M. G. A.; RIBAS, R. P.; REIS, A.I. Functional composition: A new paradigm for performing logic synthesis. **Thirteenth International Symposium on Quality Electronic Design (ISQED)**, Santa Clara, CA, 2012, pp. 236-242.

MEENA, J.S.; SZE, S.M.; CHAND, U.; TSENG, T.Y. Overview of emerging nonvolatile memory technologies. **Nanoscale research letters**, v. 9, n. 1, pp. 1-33. 2014.

MODI, N.; CORTADELLA, J. Boolean decomposition using two-literal divisors. **17th International Conference on VLSI Design. Proceedings.,** 2004, pp. 765-768.

NEUTZLING, A.; MARTINS, M.G.A.; RIBAS, R.P.; REIS, A. I. A constructive approach for threshold logic circuit synthesis. **2014 IEEE International Symposium on Circuits and Systems (ISCAS)**, Melbourne VIC, 2014, pp. 385-388.

NEUTZLING, A.; MATTOS, J.M.; REIS, A.I.; RIBAS, R.P.; MISCHENKO, A. Threshold Logic Synthesis Based on Cut Pruning. **In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**, pp. 494-499, 2015.

PALANISWAMY, A.K.; TRAGOUDAS, S. Improved threshold logic synthesis using implicant-implicit algorithms. **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, v. 10, n. 3, p. 21, 2014.

PAN, F.; GAO, S.; CHEN, C.; SONG, C.; ZENG, F. Recent progress in resistive random access memories: materials, switching mechanisms, and performance. **Materials Science and Engineering: R: Reports**, v. 83, p. 1-59, 2014.

PAPANDROULIDAKIS , G.; VOURKAS, I.; ABUSLEME, A.; SIRAKOULIS, G. C.; RUBIO, A. Crossbar-Based Memristive Logic-in-Memory Architecture. *IEEE Transactions on Nanotechnology*, v. 16, n. 3, p. 491-501, 2017.

POIKONEN, J. H.; LEHTONEN, E.; LAIHO, M. On synthesis of Boolean expressions for memristive devices using sequential implication logic. **IEEE Trans. on Computer-Aided Design on Integrated Circuits and Systems**, v. 31, n. 7, p. 1129-1134, July 2012.

Possani, V. N.; CALLEGARO, V.; REIS, A. I.; RIBAS, R. P. MARQUES, F.; da ROSA, L. S. Graph-Based Transistor Network Generation Method for Supergate Design. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 24, n. 2, p. 692-705, 2016.

RAGHUVANSHI, A.; PERKOWSKI, M. Logic synthesis and a generalized notation for memristor-realized material implication gates. **in Proc. Int’l Con. on Computer-Aided Design (ICCAD)**, pp. 470-477, 2014.

RAHMAN, K. C.; HAMMERSTROM, D.; LI, Y.; CASTAGNARO, H.; PERKOWSKI, M. A. Methodology and Design of a Massively Parallel Memristive Stateful IMPLY Logic-Based Reconfigurable Architecture. *IEEE Transactions on Nanotechnology*, v. 15, n. 4, p. 675-686, 2016.

RUDELL, R.L. **LOGIC SYNTHESIS FOR VLSI DESIGN**. Phd Thesis, University of Califronia, Berkeley, 1989.

SASAO, T. **Switching Theory for Logic Synthesis.** MA, Norwell:Kluwer, 1999.

SASO, T.; BUTLER, J. T. Worst and best irredundant sum-of-products expressions. *IEEE Transactions on Computers*, v. 50, n. 9, p. 935-948, 2001.

SHIN, S.; KIM, K.; KANG, S.-M. S. Reconfigurable stateful NOR gate for large-scale logic-array integrations. **IEEE Trans. on Circuits and Systems II: Express Briefs**, v. 58, n. 7, p. 442-446, July 2011.

SHIRINZADEH, S.; SOEKEN, M.; GAILLARDON, P. E.; DRECHSLER, R. Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs. **2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)**, Dresden, 2016, pp. 948-953.

SIEMON, A.; MENZEL, S.; WASER, R.; LINN, E. A Complementary Resistive Switch-Based Crossbar Array Adder. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, v. 5, n. 1, p. 64-74, March 2015.

SIMMONS, J.G.; VERDERBER, R. R. New conduction and reversible memory phenomena in thin insulating films. **Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences,** v. 301, n, 1464, 1967.

SOEKEN, M.; SHIRINZADEH, S.; GAILLARDON, P. E.; AMARÚ, L. G.; DRECHSLER, R., DE MICHELI, G. An MIG-based compiler for programmable logic-in-memory architectures. **2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)**, Austin, TX, 2016, pp. 1-6.

STRUJOV, D. B.; SNIDER, G. S.; STEWART, D. R.; WILLIAMS, R. S. The missing memristor found. **Nature**, v. 453, n. 7191, p. 80-83, May 2008.

SUN, X.; LI, G., DING, L.; YANG, N.; ZHANG, W. Unipolar memristors enable "stateful" logic operations via material implication. **Applied Physics Letters**, v. 99, n. 7, 2011.

TALATI, N.; GUPTA, S.; MANE, P.; KVATINSKY, S. Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC). **IEEE Transactions on Nanotechnology**, v. 15, n. 4, p. 635-650, July 2016.

TEODOROVIC, P.; DAUTOVIC, S.; MALBASA, V. Recursive Boolean formula minimization algorithms for implication logic. **IEEE Trans. on Computer-Aided Design on Integrated Circuits and Systems**, v. 32, n. 11, p. 1829-1833, Nov. 2013.

VALOV, I.; KOZICKI, M.N. Cation-based resistance change memory. **Journal of Physics D: Applied Physics**, v. 46, n. 7, p. 074005-074018, 2013.

WANG, X.; TAN, R.; PERKOWSKI, M. Synthesis of memristive circuits based on stateful IMPLY gates using an evolutionary algorithm with a correction function. *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Beijing, 2016, pp. 97-102.

WONG, H.S.P.; LEE, H.Y.; YU, S.; CHEN, Y.S.; WU, Y.; LEE, B.; CHEN, P.S.; TSAI, M.J. Metal–oxide RRAM. **Proceedings of the IEEE**, v. 100, n. 6, p. 1951-1970, June 2012.

WONG, H.-S.P.; SALAHUDDIN, S. Memory leads the way to better computing. **Nature Nanotechnology**, v. 10, p. 191–194, March 2015.

WOUTERS, D.J.; WASER, R.; WUTTIG, M. Phase-Change and Redox-Based Resistive Switching Memories. **Proceedings of the IEEE**, v. 103, n. 8, p. 1274 – 1288, Aug. 2015.

XIE, L; DU NGUYEN, H. A.; TAOUIL, M.; HAMDIOUI, S.; BERTELS, K. A Mapping Methodology of Boolean Logic Circuits on Memristor Crossbar. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, 2017.

YANG, J.; JOSHUA, J.; WILLIAMS, R.S. Memristive devices in computing system: Promises and challenges. **ACM Journal on Emerging Technologies in Computing Systems (JETC)**, v. 9, n. 2, 2013.

YANG, Y.; MATHEW, J.; PONTARELLI, S.; OTTAVI, M.; PRADHAN, D. K. Complementary Resistive Switch-Based Arithmetic Logic Implementations Using Material Implication. **IEEE Transactions on Nanotechnology**, v. 15, n. 1, p. 94-108, 2016.

ZHU, X.; YANG, X.; WU. C.; XIAO, N., WU, J.; YI, X. Performing Stateful Logic on Memristor Memory. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 60, n. 10, p. 682-686, Oct. 2013.

ZHANG, Y.; SHEN, Y.; WANG, X.; GUO, Y. A Novel Design for a Memristor-Based or Gate. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 62, n. 8, p. 781-785, 2015.