Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 183–190
DOI: 10.15439/2023F4382
ISSN 2300-5963 ACSIS, Vol. 35

XOR-based decomposition and its application in memory-based and reversible logic synthesis

Tomasz Mazurkiewicz
0000-0001-7305-2379
Cyber Command
Warsaw, Poland
Email: kontakt@tomaszmazurkiewicz.pl

***Abstract*—In this research paper, we propose a novel approach to digital circuit design using XOR-based decomposition. The proposed technique utilizes XOR gates as a fundamental building block for decomposing complex Boolean functions into simpler forms, leading to more efficient and compact digital circuits. We demonstrate the effectiveness of our approach in two different contexts: memory-based logic synthesis and reversible logic synthesis. In particular, we demonstrate that the proposed technique can efficiently reduce the number of input variables, which is a crucial task when using memories in the design. Obtained results prove that the XOR-based approach can efficiently complement variable reduction and dimensionality reduction algorithms. Furthermore, we show its application in generating the XOR-AND-XOR form of a reversible function and demonstrate how to combine it with another technique, i.e., a functional decomposition for reversible logic synthesis.**

I. Introduction

LOGIC synthesis is a very important process that enables efficient design and implementation of complex digital circuits. Its importance continues to grow since the complexity and performance demands of digital circuits are still increasing. In recent years, a specific type of logic synthesis, i.e., memory-based synthesis has gained attention from researchers, leading to the development of numerous algorithms and techniques in this field, making it a promising approach for the design of modern electronic systems. Especially logic synthesis of incompletely specified Boolean functions was deeply analyzed.

This technique approaches the design of digital circuits from a different perspective than traditional logic synthesis methods. Instead of focusing solely on Boolean logic gates and their interconnections, memory-based logic synthesis incorporates memories, such as static random-access memory (SRAM) and read-only memory (ROM), as fundamental building blocks in the design process. Due to that, this approach can improve the performance, area, and power consumption of digital circuits.

Lately, a synthesis of specific functions, called index generation functions [6], [9], [13], gained significant interest due to the practical applications of their implementations in network hardware, e.g., in telecommunication and cybersecurity.

Due to the properties of index generation functions, typically fewer variables than initial N variables can be used to represent those functions. It is important, especially in memory-based logic synthesis [11], where the memory size strongly depends on the number of input variables.

In the literature, the application of linear (i.e., XOR-based) decomposition in index generation functions minimization was widely investigated. However, XOR-based logic synthesis is a relatively new approach to a digital circuit design that leverages the properties of the exclusive OR (XOR) gate as a fundamental building block. It is worth noticing that this approach can often exploit the symmetries of Boolean functions, leading to more efficient circuits.

This approach implements a function as a composition of linear and general functions. The layer of XOR gates implements the first one, while the second one is typically implemented using memory (RAM/ROM). A typical decomposition scheme is presented in Fig. 1. Variable reduction is an optional step that reduces the number of variables, i.e. it removes those variables that can be removed without loss of any information. The outputs of this algorithm become the inputs to a linear function algorithm. This algorithm finds P reduction equations that use XOR combinations of subsets of the input variables. In the end, the general function is implemented using $2^P Q$ memory bits, where Q denotes the number of output variables.

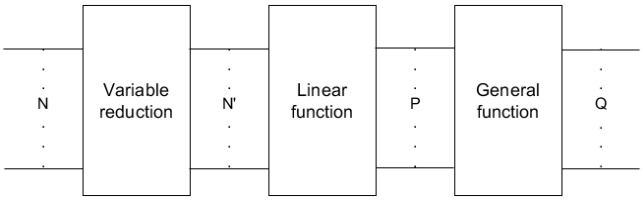

Fig. 1: The linear decomposition scheme.

Similar approaches, i.e. reduction of a number of variables (or dimensionality reduction) and implementation of a reduced general function, are used in other fields, e.g. in reversible logic synthesis [2] and data mining [3]. Especially the first field is a promising research area due to its potential to improve the energy efficiency of digital circuits. Work on the potential of the exposition of an XOR relationship in the logic synthesis of boolean functions has also been carried out. For example, Czajkowski and Brown [5] showed that it leads to significant resource savings for MCNC (Microelectronics Center of North Carolina) benchmark functions.

In this paper, we analyze how the XOR-based method used

previously in index generation functions minimization [6] can be generalized and applied to any function represented using binary input vectors. We present the whole algorithm and show its usefulness using standard benchmark functions. We also present how the proposed approach can be used in the fields mentioned above. In particular, lately, the novel form called XORAX (XOR-AND-XOR) was proposed [2] to represent a function. This form can ease the reversible synthesis of some functions. In this paper, we show that the XOR-based decomposition can be used to perform the first step of XORAX form generation. We also prove that the proposed method can be used to improve the results obtained using a functional decomposition [10].

II. Preliminaries

A. Basic notation

Let N denote the number of input variables, and K denote the number of rows in a function truth table. Notice that $K = 2^N$ for completely specified Boolean functions.

To present an algorithm of XOR-based decomposition, we introduce a concept of discernibility set. It will be denoted as $C_{p,q}$, where p and q ($p, q \in \{1, 2, \ldots, K\}, p < q$) are indexes of vectors of $\{0,1\}^N$, such that $F(p) \neq F(q)$, where $F(p)$ is the output value for row number p in a function truth table. We define discernibility set as follows:

$$C_{p,q} = \{x \in X : x(p) \neq x(q)\}, \tag{1}$$

where X denotes the set of input variables.

In particular, $C_{p,q}$ represents input variables where vectors number p and q differ. For example, in Table I the first and third vectors differ on variable x_2. Thus, $C_{1,3} = \{x_2\}$.

TABLE I: An example function.

idx	x_1	x_2	x_3	$F(X)$
1	0	0	0	0
2	0	0	1	0
3	0	1	0	1
4	0	1	1	1
5	1	0	0	1
6	1	0	1	1
7	1	1	0	0
8	1	1	1	0

Notice that for any index generation function, the condition $F(p) \neq F(q)$ is true for every possible pair of values p and q since the output values are unique consecutive integer values. This observation simplifies computations for such functions. However, it leads to higher memory consumption since many more $C_{p,q}$ sets might be generated.

The collection of all $C_{p,q}$ will be denoted as RC, i.e.

$$RC = \{C_{p,q} : p, q \in \{1, 2, \ldots, K\}, p < q\}. \tag{2}$$

Its complement, i.e., collection of all sets that are not present in the RC, will be denoted as $COM(RC)$. Additionally, the complement limited to r-element sets will be denoted as $COM(RC^r)$. The discernibility sets for all possible values of p and q can be represented using the discernibility matrix.

Example 2.1: Consider the example function ($N = 3, O = 1, K = 8$) presented in Table I. All calculated C_{pq} sets for this function (i.e., RC) are presented in Table II. Notice that pairs of p and q values such that $F(p) = F(q)$ were omitted in the calculation (e.g. $C_{1,2}$ is not present since the value for both the first and second rows equals zero). Based on the calculated sets, we get

$$RC^1 = \{\{x_1\}, \{x_2\}\},$$

$$RC^2 = \{\{x_1, x_3\}, \{x_2, x_3\}\},$$

$$RC^3 = \emptyset.$$

Therefore, we get the following complements:

$$COM(RC^1) = \{\{x_3\}\},$$

$$COM(RC^2) = \{\{x_1, x_2\}\},$$

$$COM(RC^3) = \{\{x_1, x_2, x_3\}\}.$$

TABLE II: $C_{p,q}$ sets for the example function

p, q	$C_{p,q}$	p, q	$C_{p,q}$
1,3	$\{x_2\}$	3,7	$\{x_1\}$
1,4	$\{x_2, x_3\}$	3,8	$\{x_1, x_3\}$
1,5	$\{x_1\}$	4,7	$\{x_1, x_3\}$
1,6	$\{x_1, x_3\}$	4,8	$\{x_1\}$
2,3	$\{x_2, x_3\}$	5,7	$\{x_2\}$
2,4	$\{x_2\}$	5,8	$\{x_2, x_3\}$
2,5	$\{x_1, x_3\}$	6,7	$\{x_2, x_3\}$
2,6	$\{x_1\}$	6,8	$\{x_2\}$

Since we check whether $F(p) \neq F(q)$, the proposed approach can be applied also to functions that do not return only 0 or 1. In particular, both multiple-output functions and functions with multiple-valued output can be decomposed using the same technique as long as input vectors are represented as binary vectors.

Example 2.2: Consider the function presented in Table III. It represents the following mapping: $F : \{0,1\}^6 \rightarrow \{1, 2, 3\}$. Using the same approach as in Example 2.1, we get the RC presented in Table IV.

TABLE III: An example with multiple-valued output.

idx	x_1	x_2	x_3	x_4	x_5	x_6	$F(X)$
1	1	0	0	0	0	0	1
2	0	1	1	1	1	0	1
3	0	0	1	0	0	1	2
4	1	0	1	1	1	1	2
5	0	1	0	0	1	0	3
6	0	1	0	0	0	0	3

B. Memory-based logic synthesis

Memory-based logic synthesis plays an important role in embedded systems, which often rely heavily on memory for storing and processing data. For example, nonvolatile Read Only Memories (ROMs) are used to store some fixed data used in a design, or to implement a truth table or a state machine.

TABLE IV: $C_{p,q}$ sets for the example function

p,q	$C_{p,q}$	p,q	$C_{p,q}$
1,3	$\{x_1,x_3,x_6\}$	2,5	$\{x_3,x_4\}$
1,4	$\{x_3,x_4,x_5,x_6\}$	2,6	$\{x_3,x_4,x_5\}$
1,5	$\{x_1,x_2,x_5\}$	3,5	$\{x_2,x_3,x_5,x_6\}$
1,6	$\{x_1,x_2\}$	3,6	$\{x_2,x_3,x_6\}$
2,3	$\{x_2,x_4,x_5,x_6\}$	4,5	$\{x_1,x_2,x_3,x_4,x_6\}$
2,4	$\{x_1,x_2,x_6\}$	4,6	$\{x_1,x_2,x_3,x_4,x_5,x_6\}$

This technique helps to optimize the design of digital circuits that interface with memory.

In modern embedded systems, Field-Programmable Gate Arrays (FPGAs) are used very often due to the wide range of applications, including signal processing, video processing, networking, and data storage. The fundamental building blocks used to implement a digital circuit described typically using a hardware description language, are Look-Up Tables (LUTs). A LUT is a small, programmable memory block that can store a truth table, which implements a Boolean function. The number of inputs to a LUT can vary depending on the FPGA architecture, but in modern devices equals typically 4 to 6. Additionally, FPGAs very often contain memories.

The main goal of memory-based logic synthesis [11] is to minimize the memory usage of the design. The typical approach is to use functional decomposition to divide a function into smaller subfunctions that can be efficiently implemented using memories or LUTs. Additionally, a variable reduction [3] is often used to remove redundant information from a function. However, this technique fails for some functions [6], or the number of variables can be further minimized. For example, the XOR-based decomposition reduces the number of variables by using the additional layer of XOR gates.

C. Reversible logic synthesis

Reversible logic synthesis is a process of designing circuits that can run both forward and backwards, meaning that can perform both computation and its inverse. In order to do so, the circuit has to have the same number of input and output variables and the implemented function must be a bijection (i.e., the input values can be uniquely determined from the output values and vice versa). In this process, the main goal is to minimize the number of gates required to implement a function.

In order to unify the approach for the comparison of different reversible logic synthesis methods, several gate libraries are used. The smallest complete set of gates, proposed in [14] and called the NCT gate library, contains three gates: NOT, CNOT and Toffoli gates. Any reversible function can be implemented using a combination of those three gates.

The first gate is a one-bit gate that performs the logical negation operation. The second gate is a two-bit gate, meaning it has two input signals and two output signals, which implements the following mapping $(x,y) \rightarrow (x, x \oplus y)$, where $\oplus$ denotes the XOR function. The Toffoli gate is a three-bit gate with two control bits and a single target bit. It implements the following mapping $(x,y,z) \rightarrow (x,y,xy \oplus z)$.

The NCT library is commonly used in the design of reversible circuits due to its simplicity and efficiency. However, it can be extended to include some additional gates, such as SWAP and Fredkin gates (NCTSF library) or multiply-controlled gates (GT and GT&GF libraries). Reversible circuits are implemented as a cascade of those reversible logic gates.

There are several significant applications of reversible logic synthesis. Due to the asymptotic zero power dissipation achievable by reversible computation [1], it can be used in low-power computing, making them an attractive option for mobile devices, wearables, and Internet of Things (IoT) devices. Reversible circuits are also important in quantum computing since quantum algorithms require reversible logic gates for their implementation. Reversible logic synthesis might be also useful in optical computing and DNA computing.

III. XOR-BASED DECOMPOSITION

The existence of the XOR-based decomposition can be verified using a simple test [6], [9] : $x_i \oplus x_j$ is a decomposition function of F iff $\{x_i, x_j\} \in COM(RC^2)$. Similarly, pair of functions $x_i \oplus x_j$ and $x_j \oplus x_k$ is a decomposition function of F iff $\{x_i, x_j, x_k\} \in COM(RC^3)$, and so on.

Therefore, to find a decomposition function, we need to generate RC^r and look for a decomposition using the increasing value of r. In the result, we get:

$$F(x_1, x_2, \ldots, x_N) = G(y_1, y_2, \ldots, y_{N-1}). \tag{3}$$

The first $N - r + 1$ inputs in the G function correspond to those inputs of the F function that are not used in a decomposition function. The last $r-1$ inputs are obtained using a decomposition function, e.g. y_{N-1} might equal to $x_i \oplus x_j$ if a decomposition function for $r = 2$ is found.

Example 3.1: Consider again the function F from our previous example (Example 2.1). Notice that both $COM(RC^2)$ and $COM(RC^3)$ are not empty. Therefore, two possible decompositions are:

1) $y_1 = x_3$ and $y_2 = x_1 \oplus x_2$,
2) $y_1 = x_1 \oplus x_2$ and $y_2 = x_2 \oplus x_3$.

In the first case, we get

$$F(x_1, x_2, x_3) = G(x_3, x_1 \oplus x_2),$$

while in the second case we get

$$F(x_1, x_2, x_3) = G(x_1 \oplus x_2, x_2 \oplus x_3).$$

In both cases, the number of variables was reduced by one. Truth tables for both cases are presented in Table V. Notice that the number of rows is also reduced (from 8 to 4). The first column shows the row numbers from the original truth table to which the newly generated value corresponds.

The same approach can be applied to the function from Example 2.2. Based on the RC we know that $x_1 \oplus x_3$ is a decomposition function, since $\{x_1, x_3\} \notin RC \Rightarrow \{x_1, x_3\} \in COM(RC^2)$. Therefore, we get:

$$y_1 = x_2,\ y_2 = x_4,\ y_3 = x_5,\ y_4 = x_6,\ y_5 = x_1 \oplus x_3.$$

TABLE V: The example function after decomposition.

(a) Function after decomposition number 1.

$idxs$	y_1	y_2	$G(Y)$
1, 7	0	0	0
3, 5	0	1	1
2, 8	1	0	0
4, 6	1	1	1

(b) Function after decomposition number 2.

$idxs$	y_1	y_2	$G(Y)$
1, 8	0	0	0
2, 7	0	1	0
4, 5	1	0	1
3, 6	1	1	1

Algorithm 1 Finding decomposition

1: $labels \leftarrow [\{i\} : i \in \{1, 2, \ldots, N\}]$
2: $dm, bins \leftarrow$ Algorithm_2()
3: **while** $True$ **do**
4: $dec \leftarrow$ Algorithm_3$(dm, bins)$
5: **if** dec is $None$ **then**
6: **break**
7: **end if**
8: $dm \leftarrow$ Algorithm_4(dm, dec)
9: $labels \leftarrow$ Algorithm_5$(labels, dec)$
10: **end while**
11: **return** $labels$

The complete algorithm is presented as Algorithm 1. Firstly, Algorithm 2 is used to generate a discernibility matrix. Array *labels* is used to represent the current form of a function. Based on that the Algorithm 3 is used to find a decomposition. If a decomposition is found, we need to modify the matrix (Algorithm 4) and labels (Algorithm 5, where Δ denotes the symmetric difference of two sets).

The proposed algorithm uses the discernibility matrix to find a possible representation of an input function using XOR gates. This matrix represents all generated $C_{p,q}$ sets for all possible pairs p and q ($p < q$). In Algorithm 2 a pseudocode for matrix generation is presented. Notice that in the 5th line, we check whether the value of the function differs between rows number p and q. If so, the value of the XOR operation of those rows is added to the matrix. An added row has ones on those positions where vectors number p and q differ, i.e. the $C_{p,q}$ set. In order to reduce the computational complexity of the whole algorithm, we analyze RC^r using the increasing value of r. Therefore, the rows from the matrix are divided into bins, based on their size, i.e. bin B_i represents RC^i. Furthermore, repeating values from each bin are removed.

The process of multilevel function minimization consists of iteration of the basic decomposition steps, presented as Algorithm 3. To speed up computation, we start each iteration by checking whether $|B_a| = \binom{n}{a}$ or $|B_a| = 0$. In the first case, $RC^a = \emptyset$. Thus, it is impossible to find a decomposition for that value of a. On the other hand, in the second case $C^a = \emptyset$. Thus, we can simply return $\{x_i : i \in \{1, 2, \ldots, a\}\}$. The found decomposition (lines 7-11) is represented as a vector. Indexes of bits set to 1 in this vector represent input variables that will be used to minimize an input function. For example, $comb = (11000)$ shows that XOR of the first and second input

Algorithm 2 Generation of discernibility matrix

1: $dm \leftarrow \emptyset$
2: **for** $p \leftarrow 1$ to $K - 1$ **do**
3: **for** $q \leftarrow p + 1$ to K **do**
4: **if** $F(p) \neq F(q)$ **then**
5: $dm = dm \cup (v_p \oplus v_q)$
6: **end if**
7: **end for**
8: **end for**
9: $bins \leftarrow split\_to\_bins(dm)$
10: **return** $dm, bins$

Algorithm 3 Finding decomposition (single iteration)

1: **for** $a \leftarrow 2$ to N **do**
2: **if** $|B_a| = \binom{n}{a}$ **then**
3: **continue**
4: **else if** $|B_a| = 0$ **then**
5: **return** $\{x_i : i \in \{1, 2, \ldots, a\}\}$
6: **end if**
7: **for** $comb \in combinations(n, a)$ **do**
8: **if** $\neg\exists v \in B_a : \forall c \in comb : v(c) = 1$ **then**
9: **return** $\{x_i : i \in comb\}$
10: **end if**
11: **end for**
12: **end for**
13: **return** $None$

variables is a decomposition function of F, i.e.

$$F = G(x_3, x_4, x_5, x_1 \oplus x_2).$$

Notice that in this approach we return the first found function. Such an approach is called First-Fit [6]. Other approaches to selecting decomposition functions were proposed in the literature. However, the described one is the fastest and provides good results in terms of the solution quality (i.e., the number of variables). On the other hand, it generates slightly worse results for specific functions, e.g. *M-out-of-N* coders.

For example, if $comb = (11000)$, then we get the following content of *labels*: $[\{3\}, \{4\}, \{5\}, \{1, 2\}]$ that represents the function G mentioned above. Notice that found decomposition function (i.e., using variables x_1 and x_2) is used as the last input to the new representation, while the other input variables come before it. Therefore, the algorithm will more likely find a decomposition using input variables that have not been used in the previous iterations. In the result, the compound degree (i.e., the number of inputs to the XOR operation) of each variable y_i might be similar.

Since the number of $C_{p,q}$ sets strongly depends on the value of K, the proposed approach is very efficient especially if $K \ll 2^N$.

The described approach treats the value of a function as a single output. However, a multi-output logic function

$$F : B^N \rightarrow B^Q, \tag{4}$$

Algorithm 4 Modification of discernibility matrix

```
jdx ← 1
for idx ← 1 to N do
    if idx ∉ dec then
        new_dm[:, jdx] = dm[:, idx]
        jdx ← jdx + 1
    end if
end for
for idx ← 1 to |dec| − 1 do
    col1 ← dm[:, dec[idx]]
    col2 ← dm[:, dec[idx + 1]]
    new_dm[:, jdx] = col1 ⊕ col2
    jdx ← jdx + 1
end for
return new_dm
```

Algorithm 5 Modification of labels

```
new_labels ← ∅
for idx ← 1 to |labels| do
    if x_idx ∉ dec then
        new_labels ← new_labels ∪ labels[idx]
    end if
end for
for idx ← 1 to |dec| − 1 do
    l1 ← labels[dec[idx]]
    l2 ← labels[dec[idx + 1]]
    new_labels ← new_labels ∪ (l1Δl2)
end for
return new_labels
```

where $B = \{0, 1\}$ can be implemented using XOR-based decomposition to each output variable separately. Recall that we denote by Q the number of output variables. The proposed approach is presented as Algorithm 6. In that case, the final result is a composition of found decompositions, where each function returns a single bit value.

IV. Application in memory-based logic synthesis

In this paper, we applied the proposed approach to some well-known functions. The obtained results are presented in Table VI. The variable reduction algorithm was applied to all analyzed functions, and the total number of reducts and the size of the shortest one are both presented in the table. Each reduct was then used as an input function to our linear decomposition algorithm. For each function, we show how many decompositions with the specified value of output variables (P) were found. *xor5* function is presented here to prove that described method correctly finds decomposition with a single multi-input XOR gate.

In the table, the Δ_1 and Δ_2 columns display the reduction factor in memory usage. The reduction factor is calculated using two different scenarios. The first formula, which is calculated using the following equation:

$$\Delta_1 = 2^{N-P} \tag{5}$$

Algorithm 6 Finding decomposition for each output

```
res ← ∅
for i ← 1 to Q do
    res = res ∪ Algorithm_1(X, Y[:, i])
end for
return res
```

compares the memory usage of the implementation from an input function to the implementation using XOR-gate decomposition. The second formula, which is calculated using the following equation:

$$\Delta_2 = 2^{N'-P} \tag{6}$$

compares the memory usage obtained after variable reduction to the final implementation, where N' is the size of the shortest reduct. In both formulas, the smallest value of P obtained for a function is used. For example, $\Delta_1 = 2$ means that memory usage is halved compared to the direct implementation (i.e., without using a decomposition algorithm).

The proposed approach can not directly operate on *don't care* terms, since it relies on the XOR operation. Typically, such terms can be ignored, set to 0 or set to 1. For *add6* and *clpl* function, we set each *don't care* term to 0. For *9sym* function, we add rows to make it a complete function and analyzed all possible values of *don't care* terms.

As already mentioned, memory-based logic synthesis was often used in the implementation of index generation functions. a well-known example of such a function consists of ten 40-bit vectors [12] and a unique consecutive integer value from 1 to 10 is assigned for every vector. In this paper, we denote this function as *igf40*. Using the variable reduction algorithm [3], it is possible to find more than 2200 reducts with $N' = 4$, and more than 100k in total ($N' \in \{4, 5, 6, 7\}$). In this paper, we applied the described approach to all those functions after variable reduction. For example, in Table VIIa, we present a function after variable reduction, where $N' = 5$. In Table VIIb we present that function after linear decomposition, where $y_4 = x_4 \oplus x_{32}$ and $P = 4$. In the end, the number of variables was minimized to 4 for more than 99% of reducts.

The most striking observation to emerge from the results is that the proposed approach lead to significant memory usage minimization. It is because obtained values of P are smaller than both the number of input variables N and the sizes of the shortest reducts.

When Algorithm 1 fails for some functions, e.g. *rd53*, Algorithm 6 can be used. In that particular case, $N = 5$, $K = 32$, and $Q = 3$. Therefore, we can apply the proposed approach three times, each time focusing on a single output signal. This technique leads to decompositions with the following number of inputs: $P_1 = N = 5$, $P_2 = 4$, and $P_3 = 1$. In the end, the memory usage is minimized from $2^5 * 3 = 96$ bits to $2^5 + 2^4 + 2^1 = 50$ bits. Notice that the third function does not need to be implemented using memory, since the number of inputs after decomposition (i.e., P_3) equals 1.

TABLE VI: Experimental results

Database / function	N	K	no. of reducts	size of the shortest reduct	P	no. of decompositions	Δ_1	Δ_2
9sym	9	87 (512)	1	9	8	1	2^1	2^1
add6	12	432	1	12	10	1	2^2	2^2
br1	12	34	2	7	6	2	2^6	2^1
br2	12	35	3	8	6	3	2^6	2^2
clpl	11	20	9	8	4	5	2^7	2^4
					5	4		
igf40	40	10	100172	4	4	99552	2^{36}	0
					5	620		
house	17	232	4	8	6	2	2^9	2^2
					9	2		
kaz	22	31	5574	5	2	2	2^{20}	2^3
					3	43		
					4	677		
					5	4347		
					6	505		
xor5	5	16 (32)	1	5	1	1	2^4	2^4

TABLE VII: Decomposition of *igf40* function (single reduct).

(a) A function after variable reduction.

x_4	x_7	x_8	x_{26}	x_{32}	F
0	0	1	0	1	1
1	1	1	1	0	2
1	0	1	1	1	3
1	1	0	1	1	4
1	0	0	1	0	5
1	1	0	0	0	6
0	1	1	0	0	7
1	1	1	0	0	8
0	1	0	1	1	9
0	0	1	0	0	10

(b) A function after linear decomposition.

y_1	y_2	y_3	y_4	F'
0	1	0	1	1
1	1	1	1	2
0	1	1	0	3
1	0	1	0	4
0	0	1	1	5
1	0	0	1	6
1	1	0	0	7
1	1	0	1	8
1	0	1	1	9
0	1	0	0	10

TABLE VIII: Experimental results (the multi-output approach)

Database / function	N	M	MEM	Values of P	MEM'
adr4	8	5	1280	{8,7,6,4,2}	468
igf40 {6,16,24,31}	4	4	64	{4,3,4,3}	48
igf40 {4,7,8,26,32}	5	4	128	{1,3,3,3}	26
rd53	5	3	96	{5,4,1}	48
s27_split	7	4	512	{7,6,7,4}	336
z4	7	4	512	{7,5,4,2}	180

The size of the memory required to implement a function is calculated as

$$MEM = 2^N * Q. \tag{7}$$

Furthermore, the size of a memory required after the application of Algorithm 6 equals

$$MEM' = \sum_{i=1}^{Q} 2^{P_i}. \tag{8}$$

The experimental results obtained using the multioutput approach are presented in Table VIII. The original memory size and the memory size after decomposition, using equations (7) and (8) respectively, were both presented. The function *igf40 {6,16,24,31}* is a function *igf40* after variable reduction, where $P = Q = 4$ and the values in the brackets are indexes of input variables left after the reduction. Thus, the memory size can not be further minimized using that approach. However, the multioutput approach leads to lower memory consumption. We also present a second reduct, where $N' = 5 \neq Q$. Similar results were obtained for other analyzed functions, leading to a significant minimization of memory usage ($MEM' < MEM$).

V. Application in reversible logic synthesis

Recently, a novel three-level XOR-AND-XOR form was proposed [2] to represent autosymmetric functions. It extends a popular Exclusive Sum of Products (ESOP) form (i.e., XOR-AND form) by adding an additional XOR level to the representation. Using that form, a reversible circuit implementing a function can be easily constructed.

The first XOR level is used to compute reduction equations. In particular, it reduces the number of input variables to a f_k function using XOR gates. Therefore, this step is crucial in this technique and influences the Quantum Cost of the circuit at most. Notice that this step is analogous to the approach described earlier in this paper. It corresponds to the scheme presented in 1, where the linear function corresponds to the reduction equations and the general function corresponds to the restriction function f_k. Therefore, XOR-based decomposition can be used to find reduction equations. Recall from Section II-C that those equations can be implemented using several CNOT gates if we find a decomposition function with

the value of r limited to two.

Secondly, a f_k function is implemented using any logic minimization tool. Notice that $k = N - P$. For example, an ABC [4] tool can be used to synthesize a function. *&esop* function derives ESOP from an AND-Inverter graph AIG, and *&exorcism* performs heuristic ESOP minimization [8]. This representation can be then used to find a reversible circuit that implements a f_k.

The final step is used to perform *uncomputation*, meaning that the original value is being restored on those lines that were affected by the reduction equation. It can be achieved by adding the CNOT gates used in the first level in reverse order.

Example 5.1: Consider the following function in ESOP form:

$$ESOP(F) = x_1x_3 \oplus x_1x_4 \oplus x_2x_5 \oplus x_2x_6.$$

We apply the proposed approach to find reduction equations. In the first iteration of the algorithm, we get $x_3 \oplus x_4$ as a decomposition function (since all are smaller in lexicographic order two input functions are in RC^2). Therefore,

$$F(X) = G(x_1, x_2, x_5, x_6, x_3 \oplus x_4).$$

Applying the algorithm the second time, we get $x_5 \oplus x_6$. Since it leads to $COM(RC) = \emptyset$, the algorithm ends. In the end, we get the following reduction equations:

$$y_1 = x_1,\ y_2 = x_2,\ y_3 = x_3 \oplus x_4,\ y_4 = x_5 \oplus x_6.$$

In the result, we get the following ESOP representation of the restriction f_2 ($k = 6 - 4 = 2$):

$$ESOP(f_2) = y_1y_3 \oplus y_2y_4.$$

Thus, we get the following XORAX representation of F:

$$XORAX(F) = x_1(x_3 \oplus x_4) \oplus x_2(x_5 \oplus x_6).$$

It contains 6 literals and 2 products. Notice that the ESOP representation contains 8 literals and 4 products.

In Fig. 2 the obtained reversible circuit is presented. The first two CNOT gates are used to represent the reduction equations (i.e, $x_3 \oplus x_4$ and $x_5 \oplus x_6$). Next, two Toffoli gates represent the f_2 restriction based on its ESOP form. Finally, the last two CNOT gates recover the initial value of both x_4 and x_6 variables by applying XOR operations one more time. Four CNOT gates and two Toffoli gates are used in total. Therefore, the total Quantum Cost (QC) equals

$$QC = 4 * 1 + 2 * 5 = 14.$$

Notice that the straightforward implementation from an ESOP form requires four Toffoli gates, where each of them realizes the single product of two variables. Thus, its Quantum Cost equals 20.

The proposed approach can also be combined with a functional decomposition technique [10]. This method decomposes an input function into smaller irreversible functions that are connected, and the original function is preserved. The most important fact is that each function is smaller (in terms of the number of inputs), which makes it easier to synthesize. This technique was also used in memory-based logic synthesis to reduce memory consumption [7].

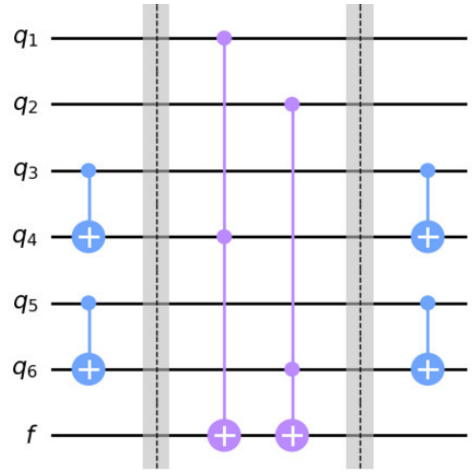

Fig. 2: Reversible circuit derived from a XORAX representation.

Each of the smaller functions is synthesized to get a reversible circuit and then combined into a single circuit. Because irreversible functions are implemented, a *garbage* output is introduced. In particular, the original value of the signal does not have to be restored on a line that was affected by reversible gates.

Example 5.2: Consider a function presented in Table IX. Using the ABC tool, we get a circuit, where Quantum Cost equals 120. Using functional decomposition it is possible, to divide this function into two functions (denoted G and H). The first function has three inputs (x_3, x_4, x_5) and one output (g). The second one has four inputs (x_1, x_2, x_6, g) and one output. The mapping between inputs and output for both functions can be found using a graph colouring method [7]. Truth tables for both functions are presented in Table X. Using the ABC tool, we get Quantum Cost 30 and 38 respectively, meaning 68 in total.

TABLE IX: An example function.

x_1	x_2	x_3	x_4	x_5	x_6	$F(X)$
0	0	0	0	0	0	1
0	1	0	0	1	0	1
0	1	0	1	1	0	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
1	0	1	1	0	0	0
0	1	0	0	0	0	0
1	0	1	0	0	0	1
0	1	0	1	0	0	0

Both functions can be decomposed using the XOR-based approach. For function G, we get $\{x_3, x_4, x_5\} \in COM(RC)$. Therefore,

$$y_1 = x_3 \oplus x_4,\ y_2 = x_4 \oplus x_5.$$

Linear function can be implemented using a single CNOT gate,

TABLE X: Functional decomposition of a function.

(a) Function G.

x_3	x_4	x_5	G
0	0	0	0
0	0	1	1
0	1	1	0
1	1	0	1
0	1	0	0
1	0	0	0

(b) Function H.

x_1	x_2	x_6	g	H
1	0	0	0	1
0	1	0	0	0
1	0	0	1	0
0	0	1	1	0
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1

while the general function synthesized using the ABC tool has $QC = 6$.

For function H, we get $\{x_2, g\} \in COM(RC)$. Therefore,

$$y'_1 = x_1,\ y'_2 = x_6,\ y'_3 = x_2 \oplus g.$$

In that case, the linear function can be implemented using two CNOT gates. The general function synthesized using the ABC tool has $QC = 5$. In total, Quantum Cost was minimized from 68 to 14.

Interestingly, this specific input function can be directly decomposed using the proposed approach. In that case, after four iterations of Algorithm 1, we get:

$$y_1 = x_4 \oplus x_6,\ y_2 = x_1 \oplus x_2 \oplus x_3 \oplus x_5.$$

Those equations can be implemented using four CNOT gates (i.e., one for y_1 and three for y_2), while the general function has $QC = 6$, leading to $QC = 10$ in total.

VI. Conclusion

In this paper, we showed how XOR-based decomposition, which has been previously used in index generation functions synthesis, can be generalized and efficiently applied in memory-based and reversible logic synthesis. We provided a complete algorithm and used well-known benchmark functions to prove that it can provide significant minimization of memory usage. The presented results highlight the potential of this technique for memory-based logic synthesis. Furthermore, we showed that our algorithm can be easily combined with other techniques proposed in the literature to achieve promising results in reversible logic synthesis. Based on our research we believe that XOR-based decomposition might become a valuable tool for other researchers and practitioners.

References

[1] Bennett C. H., "Logical Reversibility of Computation," in: *IBM Journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973, https://doi.org/10.1147/rd.176.0525.

[2] Bernasconi A., Berti A., Ciriani V., Corso G. D. and Fulginiti I., "XOR-AND-XOR Logic Forms for Autosymmetric Functions and Applications to Quantum Computing," in: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022, https://dx.doi.org/10.1109/TCAD.2022.3213214.

[3] Borowik G. and Łuba T., "Fast Algorithm of Attribute Reduction Based on the Complementation of Boolean Function," *Advanced Methods and Applications in Computational Intelligence*, 2014, pp. 25–41, https://dx.doi.org/10.1007/978-3-319-01436-4_2.

[4] Brayton, R., Mishchenko, A., "ABC: An Academic Industrial-Strength Verification Tool, " in: *Computer Aided Verification. CAV 2010. Lecture Notes in Computer Science, vol 6174*. Springer, Berlin, Heidelberg. https://dx.doi.org/10.1007/978-3-642-14295-6_5.

[5] Czajkowski T. S. and Brown S. D., "Functionally linear decomposition and synthesis of logic circuits for FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2236–2249, https://dx.doi.org/10.1109/TCAD.2008.2006144.

[6] Mazurkiewicz T. and Łuba T., "Linear and Non-linear Decomposition of Index Generation Functions," *in 26th International Conference Mixed Design of Integrated Circuits and Systems (MIXDES)*, 2019, pp. 246–251, https://dx.doi.org/10.23919/MIXDES.2019.8787031.

[7] Mazurkiewicz T., "Non-disjoint functional decomposition of index generation functions," *IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL)*, Miyazaki, Japan, 2020, pp. 137-142, https://dx.doi.org/10.1109/ISMVL49045.2020.00-16.

[8] Mishchenko A. and Perkowski M., "Fast Heuristic Minimization of Exclusive-Sums-of-Products," in: *5th International Reed-Muller Workshop*, 2001.

[9] Łuba T., Borowik G. and Jankowski C., "Gate-based decomposition of index generation functions", in: *Proc. SPIE. 10031, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016. Vol. 10031. SPIE*, 2016. pp. 100314A–1–100314A–10, https://dx.doi.org/10.1117/12.2248754.

[10] Rawski M. and Szotkowski P., "Reversible logic synthesis of boolean functions using functional decomposition," *22nd International Conference Mixed Design of Integrated Circuits & Systems (MIXDES)*, Torun, Poland, 2015, pp. 380-385, https://dx.doi.org/10.1109/MIXDES.2015.7208547.

[11] Sasao T., "Memory-Based Logic Synthesis," *Computer Science*, Springer, 2011, https://dx.doi.org/10.1007/978-1-4419-8104-2.

[12] Sasao T., "Index Generation Functions, Logic Synthesis for Pattern Matching", EPFL Workshop on Logic Synthesis & Verification, Lausanne, Switzerland, 2015.

[13] Sasao T., "Index Generation Functions," *Synthesis Lectures on Digital Circuits & Systems*, Springer Cham, 2020, https://dx.doi.org/10.1007/978-3-031-79911-2.

[14] Toffoli, T., "Reversible Computing," in: *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, Springer-Verlag, pp. 632–644, http://doi.org/10.1007/3-540-10003-2_104.