Validating two-level logic synthesis based on the Quine-McCluskey algorithm

Bochen Ye
1805673
b.ye@student.tue.nl

Zhanbo Shen
1860364
z.shen1@student.tue.nl

Jianyi Li
1824015
j.li12@student.tue.nl

Xiaoge Li
1889427
x.li1@student.tue.nl

***Abstract*—The basic rule of logic synthesis is to reduce hardware cost and delay time. To achieve this, Quine-McCluskey(QMC) algorithm is widely used for two-level logic optimization in the past time. It can simplify the input Boolean expressions on computer. It use prime implicant table to simplify the input Boolean expression which is capable of being programmed on a digital computer. This paper presents the principle of QMC algorithm and a validation on its implementation. The whole design and validation platform implemented by Python, and then it outputs validation results in terms of four aspects: consistency validation, correctness validation, running time, and gate-level circuits. Also, We compare our result of running time with related work and compare QMC algorithm with Espresso result. The experiment show that QMC algorithm is suitable for cases with few input variables, while Espresso is suitable for cases with many variables.**

***Index Terms*—Quine-McCluskey algorithm, two-level logic, logic synthesis, Validation**

I. Introduction

Logic synthesis is a process of transformation from Boolean functions to networks of logic gates. Usually, in synthesis tools, this process simplifies complex Boolean functions and then generates a netlist file, from which the corresponding gate-level logic circuits are generated. The two-level logic optimization is one of the steps in this whole process.

The task of minimizing two-logic functions is very important for logic circuit design, and for VLSI design and implementation [5]. By simplifying a Boolean function, the number of digital logic gates required to implement digital circuits can be reduced [6, 13] providing a delay-optimal two-level implementation of a logic function. The use of minimal Boolean functions can reduce the number of logic gates and might save area and power. Therefore, the optimization of two-level logic is a problem and we will address this issue in our paper.

In the past, the most common way to minimize the two-logic Boolean function is using the Karnaugh map, but the Karnaugh map can take a long time when faced with two-logic complex Boolean functions. In this case, we need to use a new algorithm called Quine-McCluskey(QMC) algorithm [8]. This algorithm is important to the analysis, design, and implementation of very large digital circuits because it can be easily implemented on a computer where can solve this problem quickly [5].

In this paper, we propose a method to validate the QMC algorithm to minimize a two-level logic Boolean function and generate a gate-level circuit base on minimization results. The procedure is conducted with a program based on the QMC algorithm to process the complex Boolean function input, and the output is minimal Boolean functions. This minimal Boolean function is used to generate the corresponding gate-level circuit. In validation part, we use Espresso logic minimizer to validate the correctness of the QMC algorithm and compare the running time of QMC algorithm with main related work.

The design approch is Python-base and allows the automatic synthesis of logic-gate form Boolean function delivering a correct-by design solution. The rest of the paper is organized as follows: Section II states the main problem. Section III discusses the related work. Section IV presents the program to convert the minimized Boolean function into a gate-level circuit. Section V shows the validation. The conclusion is summarized in section VI.

II. Problem statement

The purpose of this research is to address the problem of validating two-level logic synthesis based on the Quine-McCluskey algorithm, with delay optimization.

In this paper, we propose a method to validate the Quine-McCluskey algorithm by comparing its output with the Espresso logic minimizer, a widely accepted heuristic technique for Boolean function minimization.

The input or specification of our validation is a complex two-level Boolean function. This Boolean function only uses three basic logical operations AND, OR, NOT, and does not contain XOR. In addition, this Boolean function is two-level logic rather than multi-level logic. In two-level logic, the Boolean function is represented by two interconnected layers of logic gates. The first layer, "product term," consists of a series of AND gates that are used to compute the individual product terms in the Boolean function. The second "sum term" layer consists of one or more OR gates, which are used to sum the product terms computed in the first layer, thus achieving a complete representation of the Boolean function.

The validation process consists of using a software program implementing the Quine-McCluskey algorithm to process complex Boolean function inputs and generate minimal Boolean functions. The output of our validation is minimal Boolean functions. In this process, we must preserving the Boolean function behavior, so we validate the correctness of the Quine-McCluskey algorithm by comparing its results with those produced by the Espresso logic minimizer, which serves as a benchmark for Boolean function minimization. The platform is in Fig.1.

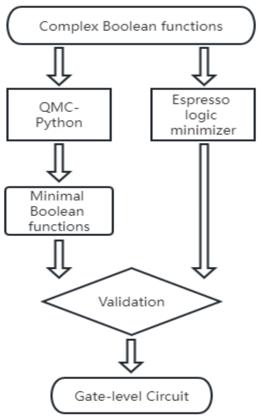


Fig. 1. The Platform

Then, we will turn the minimized Boolean function into a gate-level circuit. In this proess, we want to optimize the delay of whole gate-circuit, so we decide to put every part of the Boolean function in parallel. Beside, we do not optimize the number of gates(area) because it is a trade off between area and delay. In the end, the form of the output circuit would be an image that contain the gate-level circuit.

III. Related work

The Quine-McCluskey algorithm (abbreviated as QMC algorithm) is designed to simplify the Boolean function into the simplest form, which was developed by Willard Quine in 1952 and extended by Edward J McCluskey in 1956 [8]. Since then, much improvement has been proposed to optimize the algorithm in different domains. The basic QMC algorithm is aimed at basic gates (AND, OR & NOT). An exclusive-or logic synthesis extension of the QMC algorithm by Brian C. H. Turton was made in 1996 [12]. In 2008, T.K.Jain proposes an optimized Quine-McCluskey method that reduces the run time complexity of finding the prime implicants for minimizing Boolean expressions. Their validation compares the conventional methods of minimization and shows the advantages of the proposed method [4]. Alak Majumder derived a new approach for the minimization of Boolean functions using decimal manipulations to avoid binary comparison errors in 2015. They demonstrate superiority by comparing the performance(Smaller number of comparisons) of the improved method with the results of the base QMC algorithm [6].

While Liliya Anestieva Staneva, from Assen Zlatarov University, presents a generalized net model of the general structure for minimizing logical functions with the Quine-McCluskey algorithm, which consists of two stages: finding the full set of prime implicants and integrating them into a minimal disjunctive normal form [11]. H.A.H Mahmoud [5] has proposed a new two-level minimization method based on binary trees and it produces the minimum result in almost 99% of the experiments for minimizing logic function with less than 10 variables while time complexity is $NlogN$. Jadhav V and Buchade A. [3] propose E-sum based optimization to Quine-McCluskey Method to increase its performance by reducing the number of comparisons between minterm lists in the determination of prime implicants. Siládi V. al. [10] improved the Quine-McCluskey algorithm to run on a parallel computing system and it is implemented through CUDA.

Some other papers are about the application of the QMC algorithm. Manojlović V. [7] presents the Quine-McCluskey algorithm for minimizing switching functions, with additional specific elements, such as starting part (that is decoding DNF form) and the cost of circuit. A research group from Le Quy Don Technical University set their research direction to evaluate the performance of the QMC algorithm on multi-core CPU instead of making an improvement, which can decrease cache misses by exploiting the temporal and spatial data locality [14]. Bhandari B. al. [1] applied the Quine-McCluskey algorithm to mining the association rule from the web data which can extract the frequently accessed web pages with the minimum number of candidate sets generation.

Most of the previous validations were done by comparing running time to prove that one's proposed algorithm is better. Since we are not improving the algorithm, we validate the result of our design by comparing the run time with related work and we simply use the existing logic minimizer Espresso to prove that the results are consistent to show the correctness of our QMC algorithm.

IV. Design automation approach

When we design digital circuits, we often have to abstract them into Boolean algebraic notation. The result of a single-output circuit is specified with a Boolean function which have multi-input variables. This function is called a Boolean function or Boolean expression. Every variable only has two values: 0 or 1. This function is equal to 1 when output is present and equals to 0 when there is no output. One convenient method of specifying a Boolean function is a table of combinations such as that given in Table I. We specify that each item of the input has all the variables. In Table I, A, B, and C are the variable of a function and Y is the output for each combination.

TABLE I
Table of Combinations

Decimal	A	B	C	Y	product term
m0	0	0	0	0	$A'B'C'$
m1	0	0	1	1	$A'B'C$
m2	0	1	0	1	$A'BC'$
m3	0	1	1	1	$A'BC$
m4	1	0	0	1	$AB'C'$
m5	1	0	1	1	$AB'C$
m6	1	1	0	1	ABC'
m7	1	1	1	0	ABC

For each row of the table of combinations, a product term can be written that only equals 1 when the variables have the values listed in that row of the table. The rightmost column of Table I is called product term since any expression can be written as a sum of these product terms (SOP). We use d-entries placed on Y-column to present output which not specified for some input. It means that we not concern about the output of these inputs. It is possible to write an algebraic expression for the overall function directly from the table of combinations. This output, Y, is the sum of product terms corresponding to those rows of the table of combination for which Y is to have the value 1. See this in equation(1).

$$Y = A'B'C + A'BC' + A'BC + AB'C' + AB'C + ABC' \quad (1)$$

In Table 1, the input is four variables A, B, C, D and each variable value is 0 or 1. The decimal number corresponds to the binary variable in the leftmost column and the output Y in the rightmost column. The Boolean function of this combination table is the combination of all variable items whose output is 1. For instance, the function of Table I can be specified as $\sum m(1, 2, 3, 4, 5, 6)$.

The most basic problem is to reduce a Boolean function the simplest form from its Original expression in terms of SOP. The following part presents an algorithm for simplifying functions that can be applied to more complex functions. It is systematic and can be easily programmed on a digital computer.

A. The minimum sum

It is possible to obtain the equivalent sum function from the SOP by using the Boolean algebra function $AB + AB' = A$. Therefore, the same table of combinations could have many other same functions. For example, the function of Table I

$$Y = A'B'C + A'BC' + A'BC + AB'C' + AB'C + ABC'$$
$$= (A'B'C + A'BC) + (A'BC' + ABC') + (AB'C' + AB'C)$$
$$= (A'B'C + AB'C) + (A'BC' + A'BC) + (AB'C' + ABC')$$

can be written as either

$$Y = A'C + BC' + AB' \quad (2)$$

or

$$Y = B'C + A'B + AC' \quad (3)$$

In this paper, a literal means a variable with or without the associated prime (A, B' are literals). The sum functions which have the fewest terms of all equivalent sum functions are called minimum sums, unless these functions having the fewest terms do not all involve the same number of literals. In such cases, simplest the ones capabilities which contain the fewest literals might be known as minimal sums. For example, the function

$$Y = \sum m(7, 9, 10, 13, 14, 15)$$

can be write as either

$$Y = DBA' + CBA + DB'A + DCA'$$

or

$$Y = DBA' + CBA + DB'A + DC$$

The second expression is a minimum sum because it involves 11 literals while the first expression involves 12 literals.

B. Prime Implicants

These sections present a practical method for obtaining a minimum sum without resorting to an enumeration of all equivalent sum functions.

When the theorem $AB + A\overline{B} = A$ is used to replace two product terms which correspond to row i and row j by a single term of the table of combination, the resulting term will equal to 1 when variables have the value corresponding to either row i or row j of the table. Similarly, when this theorem continues to replace a term that equals 1 for rows k and column m by a single term, the resulting term will equal 1 for row i, j, k and m of the table of combinations. A method for obtaining a minimum sum by repeated application of this theorem ($AB + AB' = A$) was first presented by Quine [9].

In this method, the theorem is applied to all possible pairs of product terms, then to all possible pairs of the terms obtained from the product terms. It may be necessary to pair one term with several other terms in when applying this theorem. The last remaining terms are called prime implicants. Finally, a minimum sum is formed as the sum of the fewest prime implicants which when taken together will equal 1 for all required rows of the table of combinations. The terms in the minimum sum will be called minimum sum terms.

For example, the Boolean function

$$Y = \sum m(3, 7, 8, 9, 12, 13)$$

SOP expression:

$$Y = A'B'CD + A'BCD + AB'C'D' + AB'C'D + ABC'D' + ABC'D$$

The table of combination:

TABLE II
TABLE OF COMBINATIONS

Decimal	A	B	C	D	Y
m0	0	0	0	0	0
m1	0	0	0	1	0
m2	0	0	1	0	0
m3	0	0	1	1	1
m4	0	1	0	0	0
m5	0	1	0	1	0
m6	0	1	1	0	0
m7	0	1	1	1	1
m8	1	0	0	0	1
m9	1	0	0	1	1
m10	1	0	1	0	0
m11	1	0	1	1	0
m12	1	1	0	0	1
m13	1	1	0	1	1
m14	1	1	1	0	0
m15	1	1	1	1	0

The decimal numbers of Table II indicate the rows of the table of combinations for which the corresponding term will equal 1. Dashes appear in the following binary numbers, which means that this position can be either 1 or 0.

Using theorem to find prime implicants:

$$A'B'CD(0011) + A'BCD(0111) = A'CD(0-11)$$

$$AB'C'D'(1000) + AB'C'D(1001) = AB'C'(100-)$$

$$ABC'D'(100-) + ABC'D(1101) = ABC'(110-)$$

$$AB'C'(100-) + ABC'(110-) = AC'$$

In Table III, the asterisk means that this term can't be combined any more, so the prime implicants is $m(3,7)$ and $m(8,9,12,13)$ whose binary numbers is $0-11$ and $1-0-$. Therefore, the term of prime implicants is AB' and $A'CD$.

TABLE III
TABLE OF COMBINATIONS

Decimal	Binary	Minterm	Binary
m3	0011	m(3,7)	0-11*
m7	0111		
m8	1000	m(8,9)	100-
m9	1001		
m12	1100	m(12,13)	110-
m13	1101		
m(8,9)	1100	m(8,9,12,13)	1-0-*
m(12,13)	1101		

The minimum sum:

$$Y = AB' + A'CD$$

In fact, Quine's method will be more troublesome when facing Boolean functions of many variables. Fortunately, this difficulty can be overcome by converting literal into binary numbers. Using only binary numbers is more convenient for calculation, and it can be restored from binary numbers to Boolean expressions. The most important theorem being used to combine terms can be stated in terms of the binary characters as follows: First look for two items with dashes in the same position. Then if all the other items are the same except one number, we can continue to combinate these two term. If the dashes are in different positions, even if the other numbers are the same, they cannot be merged.

C. Example of Quine-McCluskey Algorithm

The QMC algorithm is based on the Quine method and extend it. In this example, the input Boolean function is in 4 variables A, B, C, D.

$$Y = \sum m(0,1,2,5,10,14)$$

This expression means that the output of the function is 1 on the combination inputs m0, m1, m2, m5, m10, and m14. The summation symbol means logical sum, which is also called logical OR.

The first step of the QMC algorithm is to find the prime implicants of the function. The prime implicants are gained by using Quine's method of the binary number. Table IV shows the table of the combination of literal.

TABLE IV
TABLE OF COMBINATIONS

Decimal	A	B	C	D	Y
m0	0	0	0	0	1
m1	0	0	0	1	1
m2	0	0	1	0	1
m3	0	0	1	1	0
m4	0	1	0	0	0
m5	0	1	0	1	1
m6	0	1	1	0	0
m7	0	1	1	1	0
m8	1	0	0	0	0
m9	1	0	0	1	0
m10	1	0	1	0	1
m11	1	0	1	1	0
m12	1	1	0	0	0
m13	1	1	0	1	0
m14	1	1	1	0	1
m15	1	1	1	1	0

The function consists of all items whose result is 1 in each row.

$$Y = A'B'C'D' + A'B'C'D + A'B'CD' + A'BC'D + AB'CD' + ABCD'$$

The minimum sum is formed by picking the fewest prime implicants whose sum will equal one for all rows of the table of combinations for which the transmission is to equal one. In terms of the characters, this means that each number in the decimal specification of the function must appear in the label of at least one character which corresponds to a minimum sum.

This SOP expression is apparently not the minimal expression. To further simplify, we need a minterm table. According to the number of 1s, the minterms are divided into different groups, see Table V.

TABLE V
TABLE OF COMBINATIONS

Number of 1s	Minterm	Binary form
Group0	m0	0000
Group1	m1	0001
	m2	0010
Group2	m5	0101
	m10	1010
Group3	m14	1110

With the minterm table, we can start to combine the minterms with each other. If two minterms differ only by one digit, that digit can be replaced by "-", which means that this digit has no influnce on the final result. For example, if the 0000 in the group0 only differs to the 0001 in the group1 in the last digit, then they can be combined as 000-. The theory for the combination is $AB + AB' = A$.

In this process, each binary character is labeled with the decimal equivalents of the binary numbers which it represents. These decimal numbers are arranged in ascending arithmetic order because when two binary numbers combine, the second number always contains all the 1's of the first number and one additional 1 so that the second number is always greater than the first. For example, $m(0,2,4,6)$ can be formed by combining $m(0,2)$ and $m(4,6)$ or by combining $m(0,4)$ and $m(2,6)$. Also, there are three ways to form characters with three dashes.

After finishing all the combinations, dividing the minterms by the numbers of 1s and form another table. The $\surd$ symbol means this term can be combined with other terms.

TABLE VI
TABLE OF COMBINATIONS

	Column I		Column II	
Number of 1s	Minterm	Binary form	Minterm	Binary form
Group0	m0	0000$\surd$	m(0,1)	000-
			m(0,2)	00-0
Group1	m1	0001$\surd$	m(1,5)	0-01
	m2	0010$\surd$	m(2,10)	-010
Group2	m5	0101$\surd$	m(10,14)	1-10
	m10	1010$\surd$		
Group3	m14	1110$\surd$		

The next step in the revised method for determining prime implicants is to list them in a column, such as that shown in Table VI and Table VII. Lines should be drawn to divide the column into groups of binary numbers that contain a given number of 1s. The theorem stated above is applied to these binary numbers by comparing each number with all the numbers of the next lower group. Other pairs of numbers do not need to be considered since any two numbers which are not from adjacent groups must differ in more than one binary digit. For each number that has 1's wherever the number (from the next upper group) with which it is being compared has 1's, a new character is formed according to the theorem. The characters in this new column(Binary form) will each contain one dash.

In Table VII, None of the items in column II can be further combined into column III, so an asterisk is used to indicate that they are prime implicants.

TABLE VII
TABLE OF COMBINATIONS

	Column II		Column III	
Number	Minterm	Binary form	Minterm	Binary form
Group0	m(0,1)	$000-*$		
	m(0,2)	$00-0*$		
Group1	m(1,5)	$0-01*$		
	m(2,10)	$-010*$		
Group2	m(10,14)	$1-10*$		

Now no terms can be combined and we get all the prime implicants, which are minterms without a $\surd$ symbol. In general, the process should be continued to size 8, size 16, etc., until no terms can be combined. In our example, the prime implicants are $m(0,1)$, $m(0,2)$, $m(1,5)$, $m(2,10)$ and $m(10,14)$. The expression can be written as

$$Y = A'B'C' + A'B'D' + A'C'D + B'CD' + ACD'$$

However, this expression might not be in the minimal format of the original. We need to find out the essential prime implicants to figure out the minimal sum of product expression.

To find the essential prime implicants, the prime implicant table is needed. Put the prime implicants in the left part of the table and put a X on the right side if the value is included in this prime implicant, but the that do not need to be considered values are not showing up in this table. We have Table VIII.

TABLE VIII
TABLE OF COMBINATIONS

		0	1	2	5	10	14
m(0,1)	A'B'C'	X	X				
m(0,2)	A'B'D'	X		X			
m(1,5)	A'C'D		X		X		
m(2,10)	B'CD'			X		X	
m(10,14)	ACD'					X	X

If the X is the only X in the column, put a $*$ on it. This indicates that $X*$ is the essential prime implicant, which should be included in the minimal sum of product expression, see Table IX.

TABLE IX
TABLE OF COMBINATIONS

		0	1	2	5	10	14
m(0,1)	A'B'C'	X	X				
m(0,2)	A'B'D'	X		X			
m(1,5)	A'C'D		X		X*		
m(2,10)	B'CD'			X		X	
m(10,14)	ACD'					X	X*

Then use a horizontal grey cell to connect every X that is in the same row with the X with an asterisk, see Table X. Also, use a vertical gray cell color to connect the X if the X is already connected in the last step, see in Table X.

TABLE X
TABLE OF COMBINATIONS

		0	1	2	5	10	14
m(0,1)	A'B'C'	X	X				
m(0,2)	A'B'D'	X		X			
m(1,5)	A'C'D		X		X*		
m(2,10)	B'CD'			X		X	
m(10,14)	ACD'					X	X*

TABLE XI
TABLE OF COMBINATIONS

		0	1	2	5	10	14
m(0,1)	A'B'C'	X	X				
m(0,2)	A'B'D'	X		X			
m(1,5)	A'C'D		X		X*		
m(2,10)	B'CD'			X		X	
m(10,14)	ACD'					X	X*

Then use yellow cell color as less as possible to connect the X that is not covered by grey color, see Table XII.

TABLE XII
TABLE OF COMBINATIONS

		0	1	2	5	10	14
m(0,1)	A'B'C'	X	X				
m(0,2)	A'B'D'	X		X			
m(1,5)	A'C'D		X		X*		
m(2,10)	B'CD'			X		X	
m(10,14)	ACD'					X	X*

The sum of the terms that are on the horizontal lines is the minimal sum of the product expression. The gray cells indicate the coverage of essential prime implicants while yellow cells indicate other coverage. According to the gray cells can be written$A'C'D$ and ACD'. Since it has to be written as a minimum Boolean function, $A'B'D'$ is chosen for the yellow coverage.

In this example, the minimal sum of product expression of the original expression

$$Y = \sum m(0,1,2,5,10,14)$$

is

$$Y_{Min} = A'C'D + ACD' + A'B'D'$$

D. Delay-optimize gate-level circuit

The result of the logic synthesis should be represented by the gate-level circuit, so there is one more synthesis between the minimal Boolean expression and the gate-level circuit.

Three gate-circuit are used in this paper, AND gate, OR gate, and NOT gate. The symbols of these gates are shown in Fig.2.

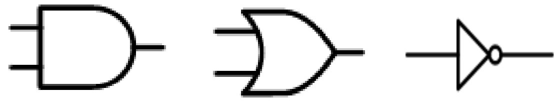


Fig. 2. The gate-level circuit

The basic mapping from the Boolean function to the gate-level circuit is added operation equals to OR gate, multiply operation equal to AND gate, and overline symbol equal to NOT gate. For example, The corresponding gate-level circuit of the minimal Boolean function $ABC + AC$ see in Fig.3.

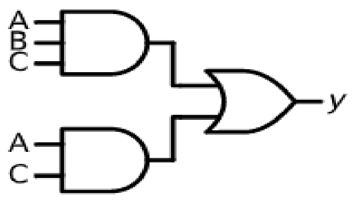


Fig. 3. The gate-level circuit

When synthesizing from Boolean expressions into gate-level circuits, one method is to generate the synthesis directly as in Fig.3, and the other is to reuse some gates as in Fig.4. Obviously the first circuit is optimized in terms of delay, only two gates have to pass from input to output, so the delay time is shorter. The second circuit reuse one gate circuit, and when the large-scale circuit is reused it reduces the total area of the circuit, which has the disadvantage that the delay time increases.

In this paper, We choose to optimize the delay time of the gate circuit rather than the area.

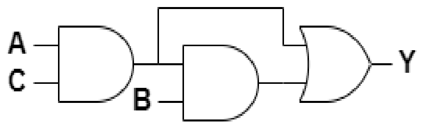


Fig. 4. The gate-level circuit(2)

V. VALIDATION

A. Implementation of design automation approach

We use Python to implement our design, so the input is a string of Boolean functions. We use & for AND gate, | for OR gate, and ∼ for NOT gate. For example, the input $\sim(A|B)\&\sim(C\&D)$ is indicate to $\overline{(A+B)}\,\overline{CD}$.

In the input two-level Boolean function, there are often $\overline{AB}$ terms that cannot be directly simplified in subsequent programs. We use De Morgan's laws to simplify $\overline{AB}$ to $\overline{A}+\overline{B}$ to make sure the result is usually SOP(sum of product). Then, this expression is expanded using the distributive law to get a Boolean function that is all of the SOP. In our design, this preprocessing function consists of two functions. The first function is *DealwithNOT* function uses De Morgan's a laws for term like $\overline{AB}$. This function uses regular expression-based functions to find items to replace (the *findall()* function in the re library) and to split (*split()* function) and combine (*join()* function) strings. Finally, the function eliminates two consecutive ∼ terms because it means that the double negation of the variable is equal to the original variable. After that, *Unfold* function use the distributive law to expand expressions. This function also uses regular expression-based functions $findall(), split(), join()$ to generate new expressions. The result of this preprocessing function is SOP so that it can be used as input to the QMC algorithm. The structure of preprocessing function is in Fig.5.

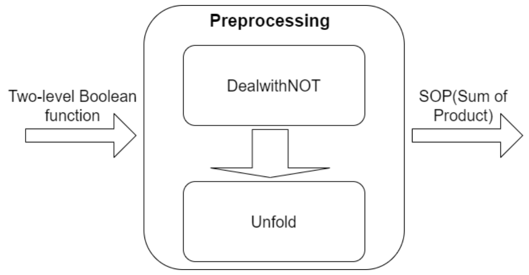


Fig. 5. The preprocessing function

In the implementation of QMC algorithm, there are three function:$Minterm(), QMC(), To\_expr()$. *Minterm(SOP, N)* function is mainly used to use the elements of $0, 1$ or $-$ to replace each item in the input SOP. $-$ stands for don't care what the specific value is. When calling this function, we need to enter not only the SOP but also the number of variables N used in this Boolean expression. The function uses the $split()$ function to split SOP according to $|$, then search the variables within each product item in order. If it is $\sim A$ then add 0 to the list, if it is A then add 1 to the list, if no corresponding variable name is found then add $-$ to the list. Finally, all sublists are combined into a complete list $minterm$. At this point, $minterm$ is a two-dimensional list, the first dimension indicates each item in the SOP form for the sum operation, and the second bit indicates each variable in each item for the product operation. The output of this function is $minterm$.

$QMC(minterm)$ function is the core of the entire implementation. The input $minterm$ is a two-dimensional list. We first traverse the entire list to remove duplicate elements. Then, we use $Combine\_terms()$ to find prime implicants. The input of $Combine\_terms()$ function is two lists consisting of $0, 1, -$ and they are compared bit by bit. There are four situations that need to be considered in $Combine\_terms()$:

- If the same value is in the same bit, then the result is also this value.
- If different value in the same bit, the the result is $-$ and record the times of different value $flag$ to make it plus one.
- If one bit is a number and the other bit is $-$, then $-$ is result, and $flag$ is recorded to make it plus one. If there are multiple occurrences in the same item and all are numbers, then it is only plus one.
- If the same bit of both items is $-$, which means that the variable at this position does not affect the values of the two logical expressions, so add $-$ to the result and record the temporary variable $temp$ to make it plus one.

At the end of the function, determining whether $flag$ is 1. if it is equal to 1, it means that only one bit of the two variables is different, and this item is a prime implicant. The $QMC(minterm)$ function is shown in Algorithm 1.

Further simplification of the obtained prime implicants is performed because there will be irrelevant terms in the existing prime implicants, i.e., the value of the term does not have an effect on the value of the whole expression. The terms in prime implicants are represented by decimal, and the number of occurrences of each decimal number is recorded by $count$. Determine the key prime implicant based on the least number of occurrences, and then select the item with the greatest coverage among the remaining prime implicants.

$To\_expr(prime\_implicants, N)$ function is opposite of *Minterm(SOP, N)* function. The input $prime_implicants$ is a two-dimensional list. The function uses $emumerate()$ to replace 1 with A and 0 with $\sim A$, and then uses & and $|$ to combine them into the final minimal Boolean expression. The structure of the QMC algorithm is in Fig.6.

```
input : minterm
output: prime_implicants
special treatment of the first line;
foreach element item of the minterm do
    remove duplicates item
end
forall $i \leftarrow minterm$ do
    special treatment of the first element of line $i$;
    forall $j \leftarrow i + 1$ do
        if items1 == items2 then
            new_item[i] =items1;
        else if items1 and items2 are digits then
            new_term[i] = '-';
        else if one of two items is '-' and another is
         '1' or '0' then
            new_term[i] = '-';
            if all '-' are in items1 or items2 then
                count the times of differences once
            else
                count the times of differences twice
            end
    end
end
if new_term exists then
    prime_implicants ← new_term ;
    if term1, term2 not in combined then
        combined ← term1, term2;
    end
end
```

Algorithm 1: The QMC algorithm

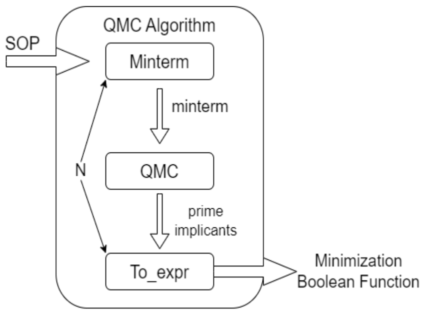


Fig. 6. The implementation of QMC algorithm

B. Validation approach

We validate our design from four aspects. The complete structure of validation in Fig.7.

- Consistency validation. The first validation is to check if the complex Boolean function(input) and minial Boolean function(output) are consistent, which means the truth tables of input and output are the same. Thus, we use a library $PyEDA$ for Python. This library can easily check the function of a Boolean expression. We use $expr()$ function to convert the input from string format to expression format of $PyEDA$ and use $equivalent()$

function to compare whether input and output are equal.

- Correctness validation. After the consistency validation, we use Espresso to get output from the same input expression. Espresso is a computer program using heuristic and specific algorithms for efficiently reducing the complexity of digital logic gate circuits [2]. Therefore, the result of Espresso is absolutely right. In order to implement, $PyEDA$ directly provides the function interface of Espresso, $espresso\_exprs()$. We can use function $espresso\_exprs()$ to get the result after $expr()$.
- Running time. As related work in Part III, we measure the running time of both two programs and compare them with the paper in III. However, the problem is running time is depend on the performance of computer, thus it is normal that the results between our program and paper will differ a lot. We use $time$ library for python and $perf\_counter()$ to implement it. The unit of time is ms.
- Gate-circuit. In order to display the results more intuitively, we show Boolean functions in the form of gate-level circuits. The library $schemdraw$ for python have the symbol of gate-circuit, so we only need to convert the format of QMC output as input of $logicparse()$ function to generate picture of gate-circuit.

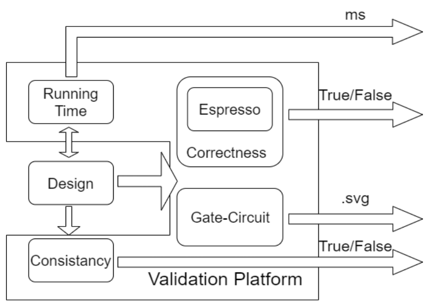


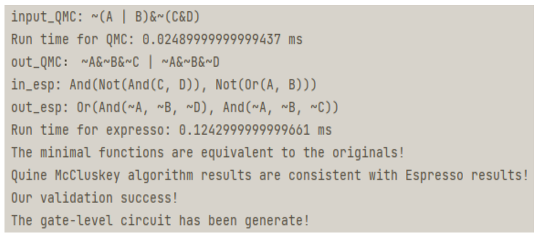
Fig. 7. The complete structure of validation

C. Validation result

The result of Consistency and Correctness validation is only a message to report if the validation is a success and the total validation success will be reported only when both validations are successful. At the same time, the running time of the QMC algorithm and Espresso will be reported. In the end, svg file is generated to show the gate-level circuit.

There is an example, we input the Boolean expression $\sim(A|B)\&\sim(C\&D)$. The validation result is in Fig.8. Besides the validation message, we also print the input expression and output expression of two programs. The gate-level circuit in Fig.9.

For running time, we compare our running time with paper [4] in Table I. The paper's algorithm was run on a machine with a 3.0 GHz processor and 512 DDR RAM and the operating system was Microsoft Windows XP Professional Edition. Our algorithm was run on a machine with a 4.2 GHz processor and 32 DDR5 RAM and the operating system was Microsoft Windows 11. We control the input variables from 7 to 11 and test the running time separately. The most important thing is that we are unable to know what the original input of the paper is, so the comparison is not very meaningful.



```
input_QMC: ~(A | B)&~(C&D)
Run time for QMC: 0.02489999999999437 ms
out_QMC:  ~A&~B&~C | ~A&~B&~D
in_esp: And(Not(And(C, D)), Not(Or(A, B)))
out_esp: Or(And(~A, ~B, ~D), And(~A, ~B, ~C))
Run time for expresso: 0.1242999999999661 ms
The minimal functions are equivalent to the originals!
Quine McCluskey algorithm results are consistent with Espresso results!
Our validation success!
The gate-level circuit has been generate!
```

Fig. 8. The report message of validation

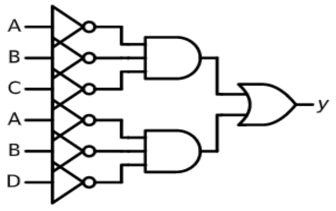


Fig. 9. The gate-level circuit output

The current QMC algorithm implementations use Petrick's method for determining all minimum sum-of-products solutions from a prime implicant chart, while we use our own algorithm which can cover most of case. Maybe this is the reason why our running time does not increase exponentially as in paper [4].

TABLE XIII
RUNNING TIME(MS)

Number of variable	QMC	Espresso	Paper's QMC
7	0.216	0.229	31
8	0.232	0.243	94
9	0.302	0.276	546
10	0.404	0.321	4162
11	0.641	0.356	34796

We use a Boolean expression with 12 product terms as input and a number of variables from 7 to 11 to record the running time. We can find that the QMC algorithm is faster when there are fewer variables, while Espresso's running time does not increase as fast when there are more variables. In Table II, we fix the 11 input variables and change product terms from 7-11. The same conclusion can be reached.

TABLE XIV
RUNNING TIME(MS)

Number of term	QMC	Espresso
7	0.213	0.238
8	0.207	0.225
9	0.264	0.290
10	0.316	0.260
11	0.387	0.292

VI. Conclusion

In this paper a method has been validated for writing any transmission as a minimum sum. A Boolean function is specified as minterm in terms of binary number and synthesis to gate-level circuit while optimizing the delay time.

This paper completely validates the feasibility of the Quine-McCluskey(QMC) algorithm for two-level logic optimization. Since the procedure for determining the prime implicants is completely systematic it is capable of being programmed on a digital computer. This validation based on Python and its library and validate QMC algorithm from four aspects. Consistency validation make sure that output are equivalent to the input. Correctness validation verifies that the QMC results are correct by comparing them with Espresso results. Validating running time by comparing results with previous papers. The output gate-level circuit allows visual validation of the relationship between the minimal expression and the circuit. Also, the experiment between QMC algorithm and Espresso show that QMC algorithm runs faster when there are fewer variables or terms as input Boolean expression.

References

[1] Bina Bhandari, RH Goudar, and Kaushal Kumar. “Quine-mccluskey: a novel concept for mining the frequency patterns from web data”. In: *International Journal of Education and Management Engineering* 8.1 (2018), pp. 40–47.

[2] Brian Holdsworth and Clive Woods. *Digital logic design*. Elsevier, 2002.

[3] Vitthal Jadhav and Amar Buchade. “Modified Quine-McCluskey Method”. In: *CoRR* abs/1203.2289 (2012). arXiv: 1203.2289. URL: http://arxiv.org/abs/1203.2289.

[4] Tarun Kumar Jain, D. S. Kushwaha, and A. K. Misra. “Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions”. In: *Fourth International Conference on Autonomic and Autonomous Systems (ICAS’08)*. 2008, pp. 165–168. DOI: 10.1109/ICAS.2008.11.

[5] H.A.H. Mahmoud. “A new method for two-level logic minimization”. In: *2003 46th Midwest Symposium on Circuits and Systems*. Vol. 3. 2003, 1405–1408 Vol. 3. DOI: 10.1109/MWSCAS.2003.1562558.

[6] Alak Majumder et al. “Investigation on Quine Mc-Cluskey method: A decimal manipulation based novel approach for the minimization of Boolean function”. In: *2015 International Conference on Electronic Design, Computer Networks Automated Verification (EDCAV)*. 2015, pp. 18–22. DOI: 10.1109/EDCAV.2015.7060531.

[7] Vladislav Manojlovic. “Minimization of Switching Functions using Quine-McCluskey Method”. In: *International Journal of Computer Applications* 82.4 (2013).

[8] E. J. McCluskey. “Minimization of Boolean functions”. In: *The Bell System Technical Journal* 35.6 (1956), pp. 1417–1444. DOI: 10.1002/j.1538-7305.1956.tb03835.x.

[9] Willard V Quine. “The problem of simplifying truth functions”. In: *The American mathematical monthly* 59.8 (1952), pp. 521–531.

[10] Vladimír Siládi et al. “Adapted parallel quine-McCluskey algorithm using GPGPU”. In: *2017 IEEE 14th International Scientific Conference on Informatics*. 2017, pp. 327–331. DOI: 10.1109/INFORMATICS.2017.8327269.

[11] Liliya Anestieva Staneva. “Minimising using the Method of Quine-McCluskey with Generalised Nets”. In: *2019 29th Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE)*. 2019, pp. 1–3. DOI: 10.1109/EAEEIE46886.2019.9000462.

[12] B.C.H. Turton. “Extending Quine-McCluskey for Exclusive-Or logic synthesis”. In: *IEEE Transactions on Education* 39.1 (1996), pp. 81–85. DOI: 10.1109/13.485236.

[13] H.J.M. Veendrick. “Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits”. In: *IEEE Journal of Solid-State Circuits* 19.4 (1984), pp. 468–473. DOI: 10.1109/JSSC.1984.1052168.

[14] Hoang-Gia Vu et al. “Performance Evaluation of Quine-McCluskey Method on Multi-core CPU”. In: *2021 8th NAFOSTED Conference on Information and Computer Science (NICS)*. 2021, pp. 60–64. DOI: 10.1109/NICS54270.2021.9701506.