Optimal Boolean Function Simplification through K-Map using Object-Oriented Algorithm

Arunachalam Solairaju[1], Rajupillai Periyasamy[2],
[1]Jamal Mohamed College, Tiruchirappalli, India. [2]Nehru Memorial College, Puthanampatti ,India

ABSTRACT
It is well known that the Karnaugh-map technique is an elegant teaching resource for academics and a systematic and powerful tool for a digital designer in minimizing low order Boolean functions. Why is the minimization of the Boolean expression needed? By simplifying the logic function, we can reduce the original number of digital components (gates) required to implement digital circuits. Therefore, by reducing the number of gates, the chip size and the cost will be reduced and the computing speed will be increased. The K-map technique was proposed by M. Karnaugh . Later Quine and McCluskey reported tabular algorithmic techniques for the optimal Boolean function minimization. Almost all techniques have been embedded into many computer aided design packages and in all the logic design university textbooks. In the present work, a well known modeling language, the object oriented technique is used for designing an Object-oriented model for Karnaugh map with the help of digital gates. An Object-oriented algorithm is also proposed for simplification of boolean functions through K-map. The Unified Modeling Language stereotypes and class diagrams are presented and performance of Unified Modeling Language model is analyzed.

Key words: Object-oriented Model. Index Terms, Karnaugh map ,Boolean functions, Mi*nterm, , Boolean functions, symbolic manipulation, binary decision diagrams, logic design verification*

1. INTRODUCTION
Boolean Algebra forms a cornerstone of computer science and digital system design. Many problems in digital logic design and testing, artificial intelligence, and combinatory can be expressed as a sequence of operations on Boolean functions. Such applications would benefit from efficient algorithms for representing and manipulating Boolean functions symbolically. Unfortunately, many of the tasks one would like to perform with Boolean functions, such as testing whether there exists any assignment of input variables such that a given Boolean expression evaluates to 1 (satisfiability), or two Boolean expressions denote the same function (equivalence) require solutions to NP-Complete or coNP-Complete problems [3]. Consequently, all known approaches to performing these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. This makes it difficult to compare the relative efficiencies of different approaches to representing and manipulating Boolean functions. In the worst case, all known approaches perform as poorly as the naive approach of representing functions by their truth tables and defining all of the desired operations in terms of their effect on truth table entries. In practice, by utilizing more clever representations and manipulation algorithms, we can often avoid these exponential computations.

Preliminaries
A variety of methods have been developed for representing and manipulating Boolean functions. Those based on classical representations such as truth tables, Karnaugh maps, or canonical sum-of-products form [4] are quite impractical---*every* function of *n* arguments has a representation of size 2*n* or more. More practical approaches utilize representations that at least for many functions, are not of exponential size. Example representations include as a reduced sum of products [4], (or equivalently as sets of prime cubes [5]) and factored into unate functions [6]. These representations suffer from several drawbacks. First, certain common functions still require representations of exponential size. For example, the even and odd parity functions serve as worst-case examples in all of these representations. Second, while a certain function may have a reasonable representation, performing a simple operation such as complementation could yield a function with an exponential representation. Finally, none of these representations are *canonical forms*, i.e. a given function may have many different representations. Consequently, testing for equivalence or satisfiability can be quite difficult. Due to these characteristics, most programs that process a sequence of operations on Boolean functions have rather erratic behavior. They proceed at a reasonable pace, but then suddenly "blow up", either running out of storage or failing to complete an operation in a reasonable amount of time.

The digital gates (Logic gates) are basic electronic components of any digital circuit. A logic gate performs a logical operation based on one or more inputs and produces a single output voltage value (i.e. voltage levels high and low). Logically these voltage values can be referred to as 1s and 0s and are used in designing and analyzing the operations of logic gates. A logic gate represents a boolean function. A boolean function is an algebraic expression formed with boolean variables (having values true or 1 and false or 0) and the logical operators (OR, AND, and NOT). There may be a large number of boolean algebraic expressions that specify a given Boolean function. It is therefore important to find the simplest representing all the input-output relationships of a digital circuit. It displays all possible input values combinations with their respective output values. There is much literature available on the concepts of digital circuits design. The basic concepts of digital circuit design are available in in many papers. Normally a Boolean expression can be given using two forms:

1. 1 Sum-of-Products (SOP): This is the more common form of Boolean expressions. The expressions are implemented as AND gates (products) feeding a single OR gate (sum).
1. 2 Product-of-Sums (POS): This is less commonly used form of Boolean expressions. The expressions are implemented as OR gates (sums) feeding into a single AND gate (product).

SOP Boolean expressions may be generated from truth tables quite easily by forming an OR of the ANDs of all input variables (standard products or minterms) for which the output is 1. POS expressions are based on the 0s, in a truth table and generated oppositely as SOP by taking an AND of the ORs of all input variables (standard sums or maxterms).

The Unified Modeling Language was created as a result of unification of different Object-oriented design methodologies. The standards and recent developments of UML are available on [8]. The good descriptions of UML diagrams and notations are available in [9] and [10]. Originally it is defined and has been successfully applied in software systems design, but can also be applied in the design of hardware systems as well. The Object-oriented design using UML diagrams in hardware system modeling and designing have been proposed in some research papers, but there is very less work available on the applications of UML in digital logic minimization. The use of UML in real-time and embedded systems specification and design has been explored by Gomaa [11] and Schattkowsky [12]. Recently Saxena et al. [13] proposed the UML model for the Multiplex system for the processes which are executing in distributed environment. Damasevicius and Stuikys [14] one. A boolean function can be represented by a truth table. A truth table is a tabular arrangement of presented a design process model for adopting Object-oriented design concepts in hardware design processes.

In the paper [21], a new class of algorithms for manipulating Boolean functions represented as directed acyclic graphs. However, we place further restrictions on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner.

Al-Rababah [15] introduced a novel approach for the synthesis of reconfigurable hardware from UML models. The presented approach enables the synthesis of Object-oriented specifications into hardware circuits. Kohut et al. [16] suggested a new approach for modeling of boolean neural networks on field programmable gate arrays using UML. Sun et al. [17] outlined a design flow to develop clocked hardware circuits using UML notations. The UML Class, Statechart and Component diagrams are used to model system specifications.

In this paper, a well known modeling language, the object oriented technique is used for designing an Object-oriented model for Karnaugh map with the help of digital gates. An Object-oriented algorithm is also proposed for simplification of boolean functions through K-map. The Unified Modeling Language stereotypes and class diagrams are presented and performance of Unified Modeling Language model is analyzed .

2. METHODOLOGY

2.1 Object Oriented Modeling and Minimization of Kmap

The UML provides the facility of defining profiles and stereotypes that can be used to define a relevant domain specific model element. The UML modeling of digital gates is shown in figure 2. In this design, a stereotype "Digital Gate" is defined. Fig.2

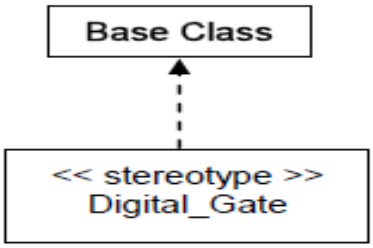


Fig. 2 Stereotype of Digital Gate

Figure 3 shows the class definition of all the digital gates.Here the class "Gate" is defined, which is derived from the stereotype "Digital_Gate". The three basic gates are defined as the subclasses of this class. The classes namely "AND", "NOT" and "OR" are inherited from the class "Gate". The other gates are defined as the classes which are the composition of these basic gates. The class "NAND" is a composition of the classes "AND' and "NOT". The class "NOR" is a composition of the classes "OR" and "NOT" where as the class "XOR" is a composition of the classes "AND", "OR" and "NOT".

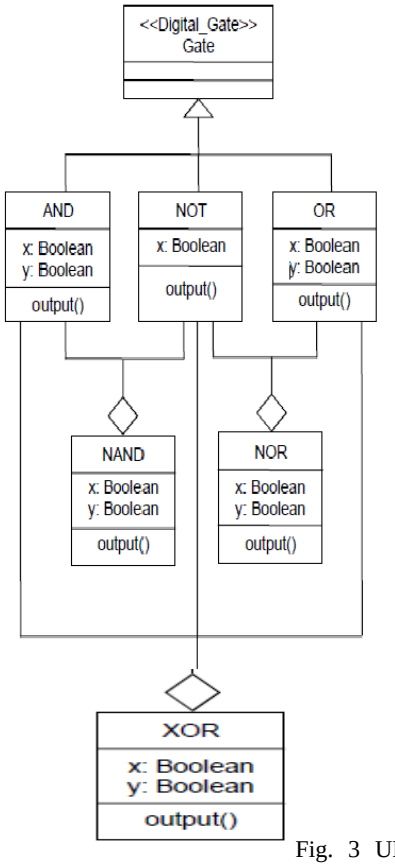


Fig. 3 UML Class Definition of Digital Gates

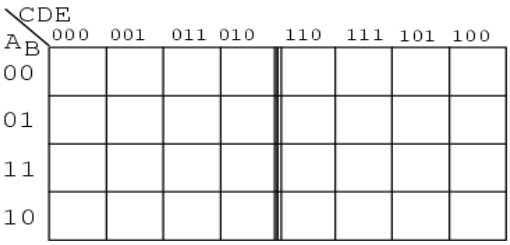
2.2 *OOP Representation and Minimization of Karnaugh Map*

The Karnaugh map is also defined using the UML stereotype mechanism. The stereotype "Map" is defined and is shown in figure 4. The class definition of Karnaugh map is shown in figure 5. In this design, a class "K-map" is defined which is derived from the stereotype "Map". This class contains multiple instances (1 to 2n) of a class "minterms" which is also defined as a stereotype. In the definition of a Karnaugh map, an attribute "n" is more significant. This attribute identifies the number of input variables. Based

on the number of inputs, the minterms are generated. The generalized instance and class instances diagram of "minterms" for 3 variables are shown in figure 6(a) and 6(b) respectively. Each minterm represents a term which is a standard product of the values of input variables. According to the K-map simplification process, these minterms are arranged in an order and they form a logical adjacency to each other. The adjacency is initialized as an attribute "adjacency" vector. The input string contains only those minterms for which the output is 1. According to the minterms specified in the input string, the objects of the class "minterms" are generated and all the attributes are initialized. For example, for input expression E = x'y'z' + x'yz + xy'z' + xy'z, the minterms inputs will be equal to $m_0 + m_3 + m_4 + m_5$. The corresponding digit values will be 000 + 011 + 100 + 101. This arrangement is shown in figure 6. The annotation mechanism of the UML is used to indicate the participating Octet, Quad or Pair. The simplification is obtained by using an algorithm described below and it is based upon the Object-oriented methodology

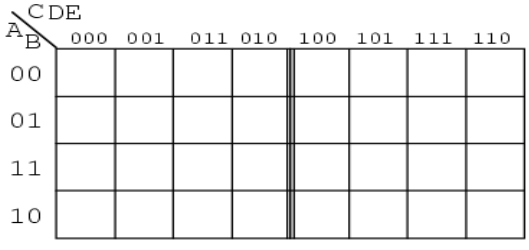
2.3 Results and Discussions

Larger Karnaugh maps reduce larger logic designs. How large is large enough? That depends on the number of inputs, *fan-ins*, to the logic circuit under consideration. One of the large programmable logic companies has an answer. Altera's own data, extracted from its library of customer designs, supports the value of heterogeneity. By examining logic cones, mapping them onto LUT-based nodes and sorting them by the number of inputs that would be best at each node, Altera found that the distribution of fan-ins was nearly flat between two and six inputs, with a nice peak at five. The answer is no more than six inputs for most all designs, and five inputs for the average logic design. The five variable Karnaugh map follows.



5- variable Karnaugh map (Gray code)

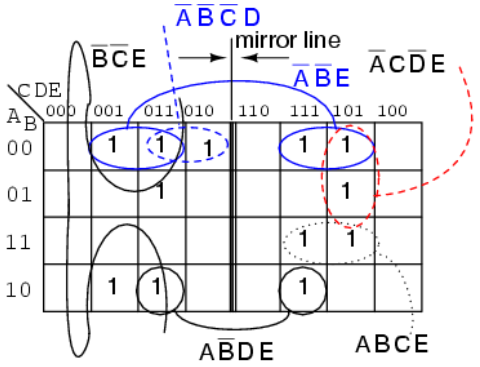
The older version of the five variable map, a Gray Code map or reflection map is shown above. The top (and side for a 6-variable map) of the map is numbered in full Gray code. The Gray code reflects about the middle of the code. The newer preferred style is below.



5- variable Karnaugh map (overlay)

The overlay version of the Karnaugh map, shown above, is simply two (four for a 6-variable map) identical maps except for the most significant bit of the 3-bit address across the top. If we look at the top of the map, we will see that the numbering is different from the previous Gray code map. If we ignore the most significant digit of the 3-digit numbers, the sequence **00, 01, 11, 10** is at the heading of both sub maps of the overlay map. The sequence of eight 3-digit numbers is not Gray code. Though the sequence of four of the least significant two bits is.

Let's put our 5-variable Karnaugh map to use. Design a circuit which has a 5-bit binary input (A, B, C, D, E), with A being the MSB (Most Significant Bit). It must produce an output logic High for any prime number detected in the input data.

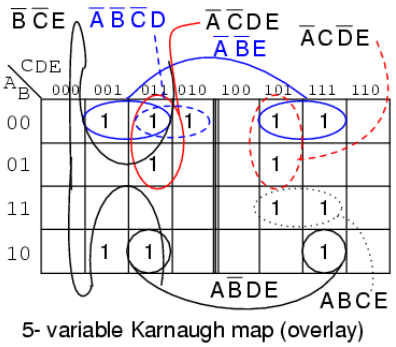


5- variable Karnaugh map (Gray code)

We show the solution above on the older Gray code (reflection) map for reference. The prime numbers are (1,2,3,5,7,11,13,17,19,23,29,31). Plot a **1** in each corresponding cell. Then, proceed with grouping of the cells. Finish by writing the simplified result. Note that 4-cell group A'B'E consists of two pairs of cell on both sides of the mirror line. The same is true of the 2-cell group AB'DE. It is a group of 2-cells by being reflected about the mirror line. When using this version of the K-map look for mirror images in the other half of the map.

Out = A'B'E + B'C'E + A'C'DE + A'CD'E + ABCE + AB'DE + A'B'C'D

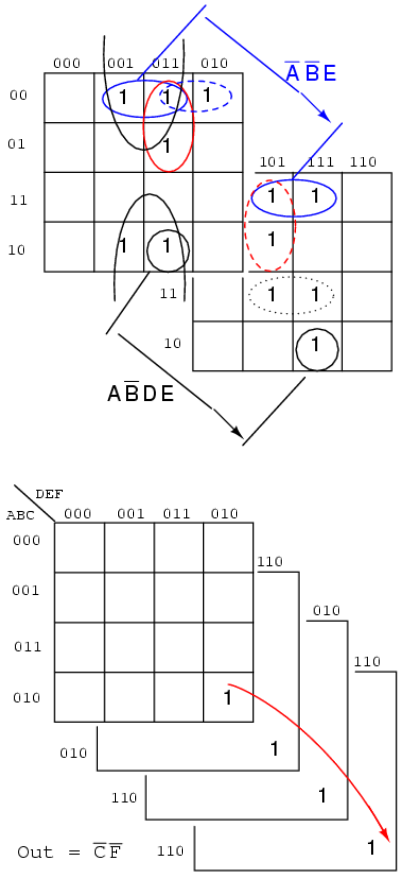
Below we show the more common version of the 5-variable map, the overlay map.



5- variable Karnaugh map (overlay)

If we compare the patterns in the two maps, some of the cells in the right half of the map is moved around since the addressing across the top of the map are different. We also need to take a different approach at spotting commonality between the two halves of the map. Overlay one half of the map atop the other half. Any overlap from the top map to the lower map is a potential group. The figure below shows that group AB'DE is composed of two stacked cells. Group A'B'E

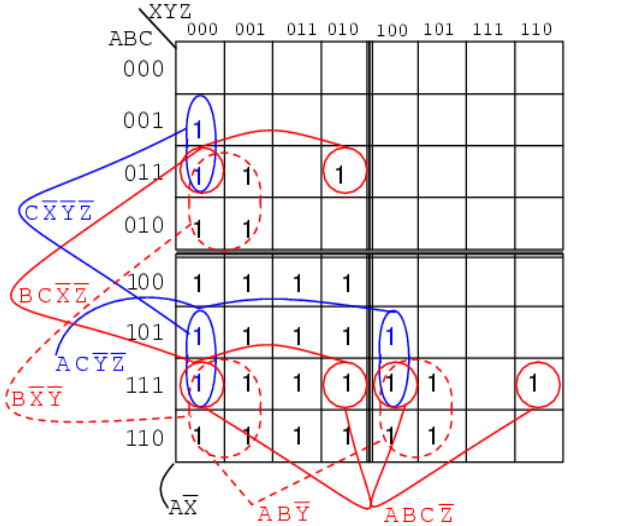
consists of two stacked pairs of cells. For the **A'B'E** group of 4-cells **ABCDE = 00xx1** for the group. That is A,B,E are the same **001** respectively for the group. And, **CD = xx** that is it varies, no commonality in **CD = xx** for the group of 4-cells. Since **ABCDE = 00xx1**, the group of 4-cells is covered by **A'B'XXE = A'B'E**.



The above 5-variable overlay map is shown stacked.

2.4 An example of a six variable Karnaugh map follows. We have mentally stacked the four sub maps to see the group of 4-cells corresponding to Out = C'F' . A magnitude comparator (used to illustrate a 6-variable K- map) compares two binary numbers, indicating if they are equal, greater than, or less than each other on three respective outputs. A three bit magnitude comparator has two inputs $A_2A_1A_0$ and $B_2B_1B_0$ an integrated circuit magnitude comparator (7485) would actually have four inputs, but, the Karnaugh map below needs to be kept to a reasonable size. We will only solve for the **A>B** output. Below, a 6-variable Karnaugh map aids simplification of the logic for a 3-bit magnitude comparator. This is an overlay type of map. The binary address code across the top and down the left side of the map is not a full 3-bit Gray code. Though the 2-bit address codes of the four sub maps is Gray code. Find redundant expressions by stacking the four sub maps atop one another . There could be cells common to all four maps, though not in the example below. It does have cells common to pairs of sub maps.

The A > B output above is ABC > XYZ on the map below.



$$Out = A\overline{X} + AB\overline{Y} + B\overline{X}\,\overline{Y} + ABC\overline{Z} + AC\overline{Y}\,\overline{Z} + BC\overline{X}\,\overline{Z} + C\overline{X}\,\overline{Y}\,\overline{Z}$$

6- variable Karnaugh map (overlay)

Wherever **ABC** is greater than **XYZ**, a **1** is plotted. In the first line **ABC=000** cannot be greater than any of the values of **XYZ**. No **1**s in this line. In the second line, **ABC=001**, only the first cell **ABCXYZ= 001000** is **ABC** greater than **XYZ**. A single **1** is entered in the first cell of the second line. The fourth line, **ABC=010**, has a pair of **1**s. The third line, **ABC=011** has three **1**s. Thus, the map is filled with **1**s in any cells where **ABC** is greater than **XXZ**. In grouping cells, form groups with adjacent sub maps if possible. All but one group of 16-cells involves cells from pairs of the sub maps. Look for the following groups:

1 group of 16-cells
2 groups of 8-cells
4 groups of 4-cells

The group of 16-cells, **AX'** occupies the entire lower right sub map; though, we don't circle it on the figure above. One group of 8-cells is composed of a group of 4-cells in the upper sub map overlaying a similar group in the lower left map. The second group of 8-cells is composed of a similar group of 4-cells in the right sub map overlaying the same group of 4-cells in the lower left map. The four groups of 4-cells are shown on the Karnaugh map above with the associated product terms. Along with the product terms for the two groups of 8-cells and the group of 16-cells, the final Sum-Of-Products reduction is shown, all seven terms. Counting the **1**s in the map, there is a total of 16+6+6=28 ones. Before the K- map logic reduction there would have been 28 product terms in our SOP output, each with 6-inputs. The karnaugh map yielded seven product terms of four or less inputs. The wiring diagram is not shown. However, here is the parts list for the 3-bit magnitude comparator for ABC>XYZ using 4 TTL logic family parts:
1 ea 7410 triple 3-input NAND gate AX', ABY', BX'Y'

2 ea 7420 dual 4-input NAND gate ABCZ', ACY'Z', BCX'Z', CX'Y'Z'
1 ea 7430 8-input NAND gate for output of 7-P-terms

3. CONCLUSION

The proposed algorithm is based on the looping of redundant terms. Therefore in order to take a closer look at how to loop two, four or eight 1's to get the least possible number of groups in a K-map table setting.Boolean algebra, Karnaugh maps, and CAD (Computer Aided Design) are methods of logic simplification. The goal of logic simplification is a minimal cost solution. A minimal cost solution is a valid logic reduction with the minimum number of gates with the minimum number of inputs. Venn diagrams allow us to visualize Boolean expressions, easing the transition to Karnaugh maps. Karnaugh map cells are organized in Gray code order so that we may visualize redundancy in Boolean expressions which results in simplification. The more common Sum-Of-Products (Sum of Minters) expressions are implemented as AND gates (products) feeding a single OR gate (sum). Sum-Of-Products expressions (AND-OR logic) are equivalent to a NAND-NAND implementation. All AND gates and OR gates are replaced by NAND gates. Less often used, Product-Of-Sums expressions are implemented as OR gates (sums) feeding into a single AND gate (product). Product-Of-Sums expressions are based on the **0**s, maxterms, in a Karnaugh map.

4. REFERENCES

1. C.Y. Lee, ''Representation of Switching Circuits by Binary-Decision Programs'', *Bell System Technical Journal*, Vol. 38, July 1959, pp. 985-999.

2. S.B. Akers, ''Binary Decision iagrams'', *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978,pp. 509-516.

3. M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

4. F.J. Hill and G.R. Peterson, *Introduction to Switching Theory and Logical Design*, Wiley, New York, 1974.

5. J.P. Roth, *Computer Logic, Testing, and Verification*, Computer Science Press, Potomac, MD., 1980.

6. R. Brayton, *et al*, ''Fast Recursive Boolean Function Manipulation'', *International Symposium on Circuits and Systems*, IEEE, Rome, Italy, May 1982, pp. 58-62.

7. B.M.E. Moret, ''Decision Trees and Diagrams'', *ACM Computing Surveys*, Vol. 14, No. 4, December 1982,pp. 593-623.

[8] OMG, UML Superstructure specification, v2.0, Retrieve from http://www.omg.org/cgi-bindoc?formal/05-07-04.

[9] Booch, G., Rumbaugh, J., Jacobson, I. (2004), The Unified Modeling Language User Guide, welfth Indian Reprint, Pearson Education.

[10] Roff, T. (2006), UML: A Beginner's Guide, Tata McGraw-Hill Edition, Fifth Reprint.

[11] Gomaa, H. (2001), Designing Concurrent, Distributed, and Real-Time Applications with UML, Proceedings of the 23rd International Conference on Software Engineering ICSE'01), IEEE Computer Society.

[12] Schattkowsky, Tim (2005), UML 2.0 - Overview and Perspectives in SoC Design, IEEE.

[13] Saxena, V., Arora D. and Ahmad S. (2007), Object Oriented Distributed Architecture System through UML, IEEE International Conference on Advanced in Computer Visio and Information Technology, ACVIT-07, Nov. 28-30, ISBN 978- 81-89866-74-7, 305-310.

[14] Kohut, R., Steinbach, B., and Fröhlich, D. (), FPGA Implementation of Boolean Neural Networks using UML.

[15] Damasevicius, R., Stuikys, V. (2004), Application of UML for Hardware Design Based on Design Process Model, IEEE.

[16] Al-Rababah Ahmad, A. (2009), UML - Models Implementations in Software Engineering System Equipments Representations, International Journal of Soft Computing Applications, Issue 4,25-34, Euro Journals Publishing, Inc.,Retrieved from http:/www. eurojournals.com / IJSCA. html

[17] Sun, Zhenxin, Wong, Weng-Fai, Zhu, Yongxin and Pilakkat, Santhosh Kumar (2005), Design of Clocked Circuits Using UML, IEEE ASP-DAC 2005 (901-904).

[18]. S. Fortune, J. Hopcroft, and E.M. Schmidt, ''The Complexity of Equivalence and Containment for Free Single Variable Program Schemes'', in *Automata, Languages, and Programming*, Goos, Hartmannis, Ausiello, and Boehm, eds., Springer-Verlag, Lecture Notes in Computer Science, Vol. 62, 1978, pp.227-240.

[19] Crenshaw, Jack W. (2003), A primer on Karnaugh maps, Embedded Systems Design, Retrieved from http://www.embedded.com/columns/programmerstoolbox / 16100908 ?Requested = 264392 .

[20] Kuphaldt, Tony R. (2007), Lessons in Electric Circuits, Volume IV – Digital, Fourth Edition, and Available as part of the Open Book Project collection retrieved from:

[21] H. Abelson, and P. Andreae, ''Information Transfer and Area-Time Trade-Offs for VLSI Multiplication '', *Communications of the ACM*, Vol. 23, No. 1, January 1980, pp. 20-23.

[22]. Vipin Saxena, Manish Shrivastava and Deepak Arora , International" Graph-Based Algorithms", Journal of Computer Science and Network Security, for Boolean Function VOL.9 No.6, June 2009