International Journal of Soft Computing 10 (2): 143-150, 2015
ISSN: 1816-9503
© Medwell Journals, 2015

A Hardware Representation for ACID Diagrams

S. Deivanai and K. Thanushkodi
Department of CSE, Akshaya College of Engineering and Technology, 642 109 Coimbatore, India

Abstract: Binary Decision Diagram (BDD) has been a popular method for representation of Boolean functions. Various methods using several variable-ordering techniques have been proposed for minimizing the size of BDD. Though, there are a number of methods to reduce the size of BDD no such data structures are introduced to challenge the BDD for decades. This study explains a novel data structure to store boolean functions called as ACID Diagram (ASCII Decision Diagram). The complete ACID diagram generation process is explained in this study with examples and several advantages of ACID diagrams are in comparison to BDDs are discussed and the complete method to implement the ACID diagram in hardware is explained. Researchers also propose several solid future works in ACID diagrams. First future research is to validate the proposed ACID diagram using several benchmark circuits and the second future research is to develop an algorithm to simplify the ACID diagrams similar to BDD simplification. The third future research is to develop methods to support don't care functions.

Key words: ACID diagrams, BDD simplification, binary decision diagrams, Boolean function manipulations, data structures

INTRODUCTION

Binary Decision Diagrams (BDD) in general is a direct acyclic graph representation of a Boolean functions proposed by Akers (1978) and Bryant (1986). The success of this technique has attracted many researchers in the area of synthesis and verification of digital VLSI circuits. Since BDDs allow efficient representation of many practical functions (Priyank, 1997; Ingo, 1987), they have became very popular data structures. The efficiency of BDDs depends mainly on the size of their graph representations.

The size of the BDD dramatically depends on the chosen order of variables (Prasad and Singh, 2003; Rudell, 1993; Ebendt, 2003). Finding a better variable order is often worth spending considerable computational effort (Aloul *et al*., 2002). Determining an optimal variable ordering is an NP-hard problem (Harlow III and Brglez, 2001). Another parameter critical during the construction of BDDs is the maximal memory requirement which is directly proportional to the number of nodes. A good ordering can lead to a smaller BDD and faster runtime whereas a bad ordering can lead to an exponential growth in the size of BDD (Aloul *et al*., 2004). Accordingly, much attention has been devoted to techniques for finding a good variable ordering. All these variable ordering techniques fall into two categories: Static Variable Ordering (SVO) algorithms (Fujita *et al*., 1988; Malik *et al*., 1988) and Dynamic Variable Ordering (DVO) algorithms (Rudell, 1993; Somenzi, 2001).

The evaluation time is also another important parameter when BDDs are used to evaluate logic functions. The evaluation time is proportional to the path length in the BDD. In general the minimization of path length in Decision Diagrams (DD) is important in databases, pattern recognition, logic simulation and software synthesis (Nagayama *et al*., 2003). The minimization of APL proposed by Nagayama *et al*. (2003), Ebendt *et al*. (2004) and Liu *et al*. (2001) reduces the average evaluation time of logic functions. The minimization of the APL leads to circuits with a smaller depth on the paths from Root to terminal nodes. By this, the circuit is optimized for speed on the one hand and on the other hand the number of very long paths is reduced (Fey *et al*., 2004). The APL minimization is very much effective in real time operating system applications (Nagayama and Sasao, 2004; Balarin *et al*., 1999; Lindgren *et al*., 2000). The minimization of LPL (Longest Path Length) of BDD can reduce the longest evaluation time which is more important for Pass Transistor Logic (PTL) (Shelar and Sapatnekar, 2001; Bertacco *et al*., 1997). So, the minimization of longest evaluation time will improve the performance of the circuit (Shelar and Sapatnekar, 2001; Bertacco *et al*., 1997).

A set of BDDs representing many functions can be combined into a graph that consists of BDDs sharing sub-graphs among them. This method saves time and space for duplicating isomorphic BDDs. By sharing all the isomorphic sub-graphs completely, no two nodes that express the same function coexist in the graph. Such a

Corresponding Author: S. Deivanai, Department of CSE, Akshaya College of Engineering and Technology, 642 109 Coimbatore, India

graph is called as Shared BDD (SBDD) or multi-rooted BDD. In a shared BDD, no two root-nodes express the same function (Raseen and Thanushkodi, 2009).

PRELIMINARIES

In the following, researchers review some of the basic definitions for BDDs.

Definition 1: A BDD is a Directed Acyclic Graph (DAG). The graph has two sink nodes labeled 0 and 1 representing the Boolean functions 0 and 1. Each non-sink node is labeled with a boolean variable v and has two out-edges labeled 1 (if then) and 0 (or else). Each non-sink node represents the Boolean function corresponding to its 1 edge if v = 1 or the Boolean function corresponding to its 0 edge if v = 0.

Definition 2: An Ordered BDD (OBDD) is a BDD in which each variable is encountered no more than once in any path. The order of variables is same along each path.

Definition 3: A Reduced OBDD (BDD) is an OBDD that is reduced by two reduction rules: deletion rule and merging rule. These reduction rules remove redundancies from the OBDD.

The size of a BDD is largely affected (and varies from linear to exponential) by the choice of the variable ordering. Figure 1 illustrates the effect of the variable ordering on the size of BDDs, for the following Boolean Eq. 1:

$$f = x_1 \cdot x_2 + x_1 \cdot \overline{x_2} \cdot x_3 \cdot x_4 + \overline{x_1} \cdot x_3 \cdot x_4 \tag{1}$$

Definition 4: In a BDD, a sequence of edge and nodes leading from the root node to a terminal node is a path. The number of non-terminal nodes on the path is the path length.

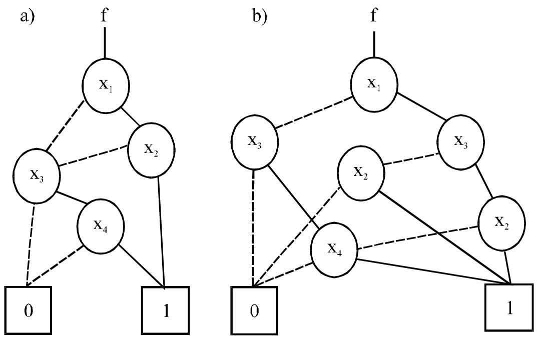

Fig. 1: Effect of variable ordering on the size of BDDs; a) x_1, x_2, x_3, x_4 and b) x_1, x_3, x_2, x_4

Definition 5: The APL is equal to the sum of the node traversing probabilities of the non-terminal nodes (Nagayama *et al*., 2003; Nagayama and Sasao, 2004) which give in the Eq. 2:

$$\mathrm{APL} = \sum_{i=0}^{N-1} P(V_i) \tag{2}$$

where, N denotes the number of non-terminal nodes.

Definition 6: The edge traversing probability, denoted by $P(e_{i0})$ or $P(e_{i1})$, is the fraction of all 2n assignments of values to the variables whose path includes (e_{i0} or e_{i1}) where e_{i0} or e_{i0} denotes the 0-edge (or the 1-edge) directed from away node V_i (Nagayama *et al*., 2003). Since, all paths include the root node, this node is traversed with probability 1.00. Since, all assignments to values of variables are equally likely, researchers can use the following Eq. 3 to calculate the $P(V_i)$ for the rest of the nodes:

$$\frac{P(V_i)}{2} = P(e_{i0}) = P(e_{i1}) \tag{3}$$

Definition 7: The Longest Path Length (LPL) of a BDD denoted by LPL (BDD) is the length of the longest path from the root to terminal node.

Definition 8: In a Decision Diagram (DD) for logic function, the memory size of the DD, denoted by Mem (DD), is the number of words needed to represent the DD in memory (Nagayama and Sasao, 2004).

In a memory, each non-terminal node requires an index and pointers to the succeeding nodes. Since, each non-terminal node in a BDD has two pointers, the memory size needed to represent a BDD is:

$$\mathrm{Mem(BDD)} = (2+1) \times \mathrm{nodes(BDD)} \tag{4}$$

Property of a single BDD node: A single BDD node is shown in Fig. 2. It can be seen from the Fig. 2 that a typical BDD node has many parents (F1, F2, ..., Fn) and two children (solid line and dashed line marked as L and R).

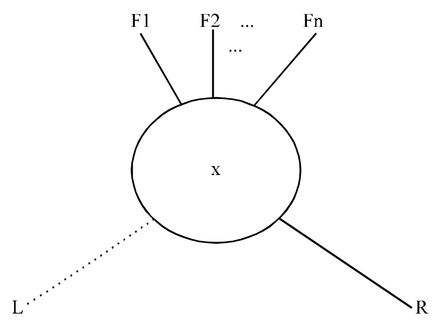

Fig. 2: A single BDD node

Proposed ACID diagram: In the proposed system called as ACID Diagram (ASCII Decision Diagram) the decision diagram is generated using the following algorithm.

Step 1: The number of input variables is split into levels of 8 input variables each. The total number of levels is given by Eq. 5:

$$NL = \left\lceil \frac{NV}{8} \right\rceil \qquad (5)$$

Where:
NV = Number of input variables
NL = Number of levels

If the number of input variables is not a multiple of 8 then the last level is formed with <8 variables. For example, if the number of variables is 21 then the 21 variables is split into 3 levels of 8, 8 and 5 variables each. It should be noted that the maximum number of variables in each level is 8 and the minimum of variables in each level is 1.

Step 2: Each node of 8 (or less) variables is expanded as a flat array of 256 (or less) items each. Apart from the 256 (or less) items, each node has a header of 4 bits. The total contents of each node is given by the Eq. 6:

$$IL = 2^{NVL} + 4 \text{ bits} \qquad (6)$$

Where:
IL = Number of items in level
NVL = Number of variables in level in the ACID diagram

For example if there are 5 bits in a level the number of items is 2^5(32) 32 plus 4 bits. A single level may also have many ACID diagram nodes. It should be noted that all the nodes in each level has the same number of variables. The reason for having the split (of bits in header) is that the header indicates the number of variables in the level and the items indicate the pointers to the next level of variables. The header may also indicate if the item is the final level by pointing to logic 0 or logic 1. The meaning of the 4 bit header is shown in Table 1.

The complete model of a single node (8 bits) of an ACID diagram node is shown in Fig. 3. From Fig. 3, it can be seen that the header has 1000 (binary) thus indicating that there are 8 bits in the node. From 8 bits, the total combination of 2^8(256) pointers is shown in the Fig. 3.

Figure 4 shows the ACID diagram node for 5 variables. The header has the value of 0101 indicating that there are 5 variables in the node. These 5 variable pointers are shown in separate 32 field values as in Fig. 4.

Table 1: Meaning of the ACID diagram node header

Header bits (binary)	Meaning
0000	Logic zero
0001	1 variable in a node
0010	2 variables in a node
0011	3 variables in a node
0100	4 variables in a node
0101	5 variables in a node
0110	6 variables in a node
0111	7 variables in a node
1000	8 variables in a node
1001-1110	Future use
1111	Logic one

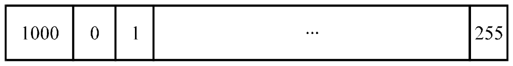

Fig. 3: A complete ACID diagram node

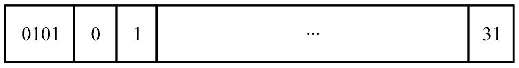

Fig. 4: An ACID diagram node with 5 variables

Fig. 5: Node 0 and 1 (ACID diagram node headers)

Figure 5 shows the ACID node for logic limiters (zero and one), respectively. It should be noted that the headers have only 0000 and 1111 (for logic zero and one) as shown in Table 1.

Step 3: The generated nodes of the ACID diagram is then linked with each other using the pointers. Since, there are eight variables in each group, there are a maximum of 265 items. Since, the value of the values range from ASCII zero to ASCII 255, the proposed method is called as ASCII decision diagram which is ACID diagrams in short.

EXAMPLE FOR GENERATING THE ACID DIAGRAM

Consider the following truth table (Table 2) with three inputs and one output. Since, there is only 3 variables the ACID diagram has only one level. The complete ACID diagram is shown in Fig. 6. The header in Fig. 6 has the value 0011 indicating that there are 3 variables. For three variables there are 8 fields. These 8 fields point to logic 0 or logic 1 (0000 or 1111 header).

For diagrams with >8 variables various levels of nodes are used. These various levels of nodes are joined together with the help of pointers.

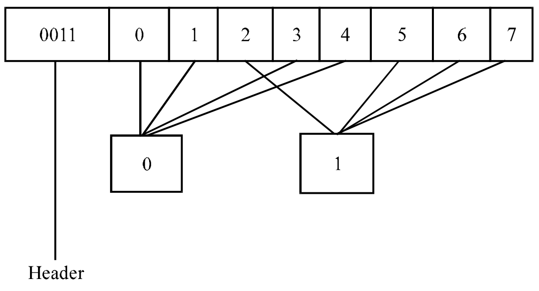

Fig. 6: ACID diagram with 3 input variables

Table 2: Truth table with 3 inputs and 1 output

Input A	Input B	Input C	Output Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

ADVANTAGES OF ACID DIAGRAMS

Access time: For BDD the total access time is given by Eq. 7:

$$\text{TTBDD} = \text{NNBDD}\times(\text{TLNODE}+\text{EPTR}) \quad (7)$$

Where:

TTBDD	= Total time for BDD operation
NNBDD	= Number of nodes in the BDD
TLNODE	= Time needed to load a node from the memory
EPTR	= Time needed to evaluate the next node pointer based on the value of the pointer

The total access time for ACID diagram is given by Eq. 8:

$$\text{TTACIDD} = \text{NLACID}\times(\text{TLLEVEL}+\text{EPTR}) \quad (8)$$

Where:

TTACID	= Total time for ACID diagram operation
NLACID	= Number of levels in the ACID diagram
TLLEVEL	= Time to load a single level
EPTR	= Time to evaluate the next pointer

Usage of processor registers directly: In BDDs the data structure is distributed, i.e., data is spread across different levels of left and right pointers. Thus, accessing these data structures is a processor intense task. Whereas, in ACID diagrams the pointers are spread as a straight array of 256 (or less) pointers. Thus, ACID diagrams can use the processor registers directly which saves a lot of resources.

No waste of memory: In BDDs to take a decision at each node the value of the variable is loaded into the memory

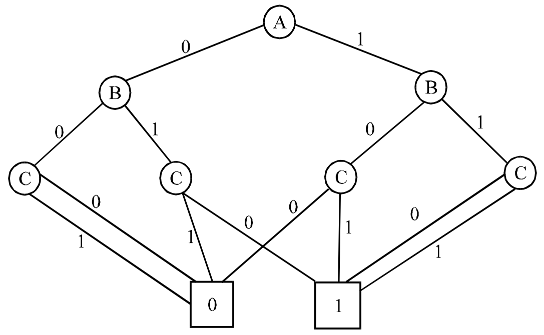

Fig. 7: BDD for 3 variables

and then compared. This is wastage of memory since the complete memory element is loaded and only one bit is accessed. In case of ACID diagrams the comparison is made with complete unit which saves a lot of memory.

Worst case number of pointers: Table 2 illustrates a Boolean function with three input variables and one output variable. The worst case un-simplified BDD for this function is shown in Fig. 7. The number of pointers used in this worst case BDD is given in Eq. 9:

$$\text{TBDDPTR} = 1+2+4+8 = 15 \quad (9)$$

where, TBDDPTR is total BDD pointers. In case of ACID Diagrams (Fig. 6) the total number pointers used is given by Eq. 10:

$$\text{TACIDPTR} = 1+8 = 9 \quad (10)$$

where, TACIDPTR is total pointers in ACID diagram (Fig. 6) pointers. Comparing Eq. 9 and 10, it can be seen that ACID diagrams uses less number of pointers and thus is much faster compared to similar BDDs.

HARDWARE REPRESENTATION OF ACID DIAGRAMS

Raseen and Thanushkodi (2009) gives the various hardware and software representation of BDD. Similar to Raseen and Thanushkodi (2009), researchers are proposing the hardware representation of ACID diagrams.

ACID diagram node black box: A typical node of an ACID diagram is shown in Fig. 8. Figure 8 is a black-box representation of an ACID diagram node. A typical ACID diagram node has two sets of inputs and one set of output. The values of the headers in the ACID diagram node are stored inside the node and does not change even if the inputs to the ACID diagram node changes. The inputs to the ACID diagram node are:

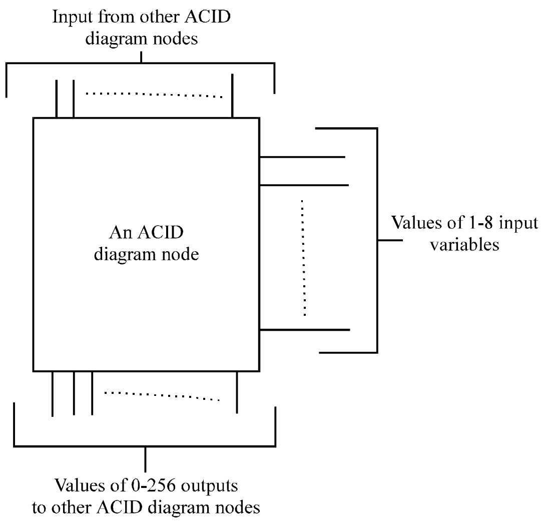
Input from other ACID diagram nodes
An ACID diagram node
Values of 1-8 input variables
Values of 0-256 outputs to other ACID diagram nodes

Fig. 8: Black box of ACID diagram node

- Binary values (1 or 0) from the parent nodes (nodes from previous levels)
- Binary values (1 or 0) from, one to eight input variables

The output of ACID diagram node is 0-256 lines. Only one 0-256 lines gets the value of logic 1 and others get the value of logic 0. This means that only one of output lines of the ACID diagram is selected and other are not.

Components of the ACID diagram node: The complete ACID diagram node can be divided into many components or parts for easier visualization of the functionality.

Inputs from previous levels: First step is to process the binary values from previous levels (higher levels). The ACID diagram node gets activated if inputs from any one of the previous level is logic 1. This can be implemented by ORing all the inputs from the previous level. This process is shown in Fig. 9. In Fig. 9 the values from the previous levels is ORed and stored in a intermediate signal ‘A’.

Values from one to eight input variables: Each ACID diagram node has inputs as 8 (or less) signals from eight input variables. Depending upon the values of the variables the intermediate signal ‘A’ is de-multiplexed to 256 (or less) lines. These intermediate lines are called as DMB (De-Multiplexed Bus). Apart from these signals the de-multiplexer has an enable line which is controlled by the signal from the ACID diagram node header. This de-multiplexer component of the ACID Diagram node is shown in Fig. 10.

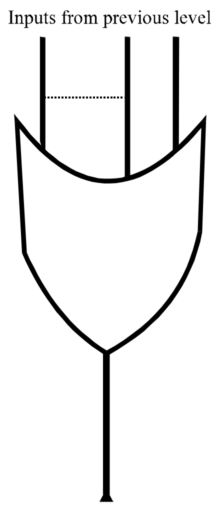

Fig. 9: Processing inputs from previous levels

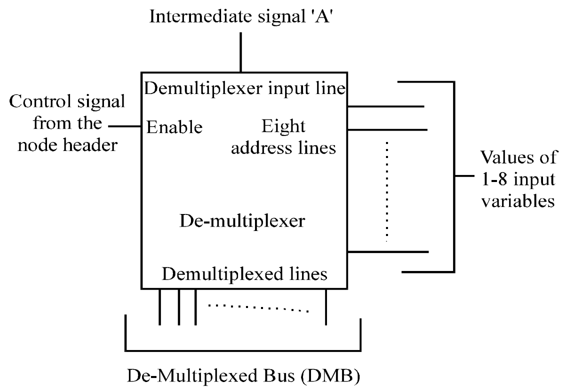

Fig. 10: De-multiplexer component of ACID diagram node

Processing of ACID diagram node header: The header of the ACID diagram node decides which output line of the ACID diagram node is active. The ACID diagram node header is of size 4 bits. These 4 bits are provided as inputs to 16 bit output de-coder. The meaning for the output of the de-coder is shown in Table 1. The realization of the de-coder circuit is shown in Fig. 11. The decoder has 4 input lines from the ACID diagram node header. It has 16 output lines. The eight output lines (corresponding to 0001-1000) go as input to the temporary Decoder BUS (DCB). The remaining eight lines go as input to the enable signal for the de-multiplexer component.

The ACID diagram output generator AND gate: The output of the ACID diagram node is generated by ANDing the individual lines from the De-Multiplexer Bus (DMB) and Decoder Bus (DCB). The width of DMB and DCB is 256 lines (from 0-255). Each and every line from the DMB in order (0-255) is ANDed with the corresponding

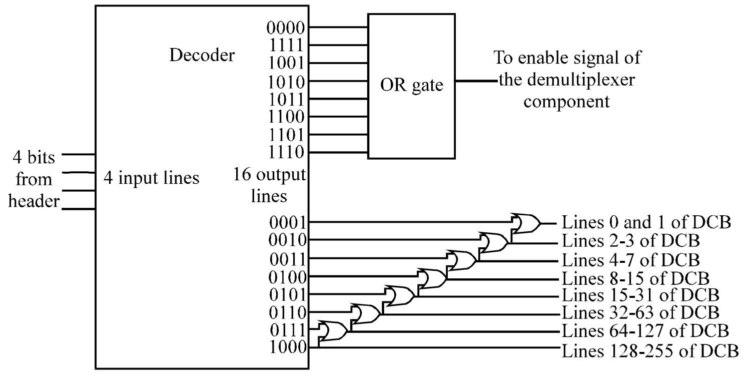

Fig. 11: The decoder component of the ACID diagram node

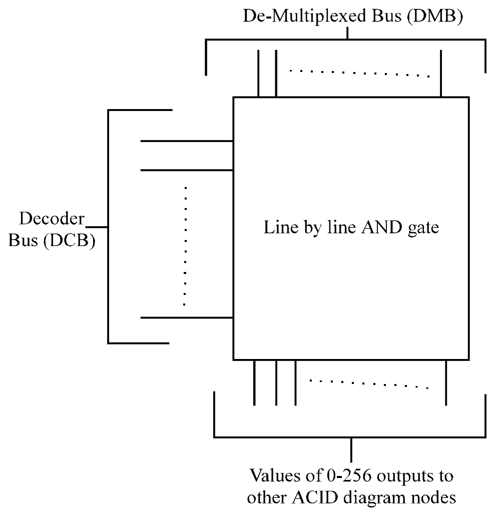

Fig. 12: ACID diagram node output generator

order (0-255) to generate the signal of the final output bus. It should be noted that only one of the final 256 signals gets the value of logic 1. If the ACID Diagram node is a terminal node (Logic 0 or logic 1) then all the output lines gets logic 0. This process is shown in Fig. 12. Figure 12 has 256 input lines from DCB and 256 input lines from DMB and 256 output lines for other ACID diagram nodes.

The complete block diagram of ACID diagram node: The complete block diagram of an ACID diagram node is shown in Fig. 13. Figure 13 has the flowing blocks:

- OR gate processor from previous node
- De-multiplexer component that handles the values from eight variables
- De-coder component that handles the values from the ACID diagram node header
- Final AND gate that generates the output from the ACID diagram node

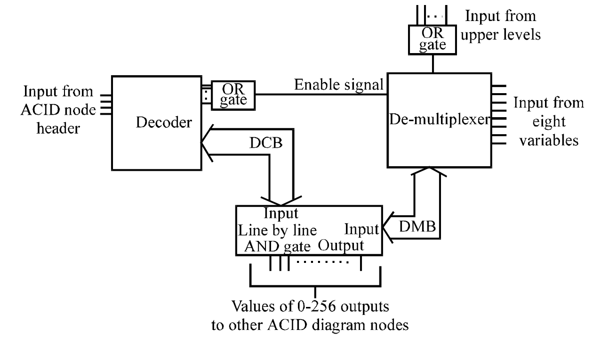

Fig. 13: Block diagram of ACID diagram node

Thus, the hardware representation of the complete ACID diagram node is explained.

CONCLUSION

In this study, researchers have proposed a novel data structure called as ACID diagrams (ASCII decision Diagrams) for storing Boolean function. The input variables are grouped into batches of eight variables each. Each group is further expanded into 256 (or less) items. Each item points to other groups of variables (or logic one or logic zero). Each group also has a header which indicates the number of variables in the group (or logic one or logic zero). The complete protocol of the group is explained in detail using diagrams and tables. Unused combinations in the header bits are reserved for future work. The construction of the ACID diagram is also explained with an example of three bit function. Advantages of ACID diagrams such as diagram access time, processor register usage, memory wastage and number of pointers used are discussed in detail in the advantages section. The complete logic circuit to implement the ACID diagram node is explained in detailed. Components of the ACID diagram node are explained. The components are then joined to build the full block diagram of the ACID diagram node. Several future research is also proposed. The first proposed future research is to build BDD and ACID diagram for benchmarks and prove that ACID diagrams are better than BDDs. Another future research of this study will be to develop algorithms for reduction of ACID diagrams. Yet, another future research will be to develop algorithms and data structures to store do not cares. The algorithms and data structures described and developed in this study will lead to more efficient protocols to store boolean functions.

RECOMMENDATIONS

Validation of proposed method using benchmarks: One of most strong future research in this study is to validate the proposed method using standard benchmarks. This process is explained as follows:

- Step 1: load the benchmarks into the memory
- Step 2: build the BDD for the benchmark function
- Step 3: build the ACID diagram for the benchmark function
- Step 3: compare the size of BDD and ACID diagram
- Step 4: from the comparison of the sizes, prove that ACID diagrams and efficient than BDDs

Reduction of ACID diagram similar to BDD: The BDD shown in Fig. 7 can be easily reduced using the standard reduction techniques. The future research proposed in this study is to develop techniques to reduce ACID diagrams.

Approach of don't cares: This study explains the approach to build and access ACID diagrams for Boolean zero and Boolean one function. Yet, another proposed research in this study is to develop ACID diagram algorithms for don't care functions.

REFERENCES

Akers, S.B., 1978. Binary decision diagrams. IEEE Trans. Comput., C-27: 509-516.

Aloul, F.A., I.L. Markov and K.A. Sakallah, 2002. Improving the efficiency of circuit-to-BDD conversion by gate and input ordering. Proceedings of the 20th International Conference on Computer Design, September 16-18, 2002, Freiburg, Germany, pp: 64-69.

Aloul, F.A., I.L. Markov and K.A. Sakallah, 2004. MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation. J. Universal Comput. Sci., 10: 1562-1596.

Balarin, F., M. Chiodo, P. Giusto, H. Hsieh and A. Jurecska *et al*., 1999. Synthesis of software programs for embedded control applications. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., 18: 834-849.

Bertacco, V., S. Minato, P. Verplaetse, L. Benini and G. De Micheli, 1997. Decision diagrams and pass transistor logic synthesis. Technical Report No. CSL-TR-97-748, Stanford University, Stanford, CA., USA., December 1997.

Bryant, R.E., 1986. Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput., C-35: 677-691.

Ebendt, R., 2003. Reducing the number of variable movements in exact BDD minimization. Proceedings of the International Symposium on Circuits and Systems, Volume 5, May 25-28, 2003, Bangkok, Thailand, pp: 605-608.

Ebendt, R., W. Gunther and R. Drechsler, 2004. Minimization of the expected path length in BDDs based on local changes. Proceedings of the Asia and South Pacific Design Automation Conference, January 27-30, 2004, Kanagawa, Japan, pp: 865-870.

Fey, G., J. Shi and R. Drechsler, 2004. BDD circuit optimization for path delay fault testability. Proceedings of the Euromicro Symposium on Digital System Design, August 31-September 3, 2004, Rennes, France, pp: 168-172.

Fujita, M., H. Fujisawa and N. Kawato, 1988. Evaluation and improvement of Boolean comparison method based on binary decision diagrams. Proceedings of the IEEE International Conference on Computer Aided Design, November 7-10, 1988, Santa Clara, CA., USA., pp: 2-5.

Harlow III, J.E. and F. Brglez, 2001. Design of experiments and evaluation of BDD ordering heuristics. Int. J. Software Tools Technol. Transfer, 3: 193-206.

Ingo, W., 1987. The Complexity of Boolean Functions. B. G. Teubner, Stuttgart, Germany, ISBN-13: 9780471915553, Pages: 457.

Lindgren, M., H. Hansson and H. Thane, 2000. Using measurements to derive the worst-case execution time. Proceedings of the 7th International Conference on Real-Time Systems and Applications, December 12-14, 2000, Cheju Island, pp: 15-22.

Liu, Y., K.H. Wang, T.T. Hwang and C.L. Liu, 2001. Binary decision diagram with minimum expected path length. Proceedings of the Conference on Design, Automation and Test in Europe, March 13-16, 2001, Munich, Germany, pp: 708-712.

Malik, S., A. Wang, R. Brayton and A. Sangiovanni-Vincentelli, 1988. Logic verification using binary decision diagrams in a logic synthesis environment. Proceedings of the International Conference on Computer Aided Design, November 7-10, 1988, Santa Clara, CA., USA., pp: 6-9.

Nagayama, S. and T. Sasao, 2004. On the minimization of longest path length for decision diagrams. Proceedings of the 13th International Workshop on Logic and Synthesis, June 2-4, 2004, Temecula, CA., USA., pp: 28-35.

Nagayama, S., A. Mishchenko, T. Sasao and J.T. Butler, 2003. Minimization of average path length in BDDs by variable reordering. Proceedings of the 12th International Workshop on Logic and Synthesis, May 28-30, 2003, Laguna Beach, CA., USA., pp: 207-213.

Prasad, P.W.C. and A.K. Singh, 2003. An efficient method for minimization of binary decision diagrams. Proceedings of 3rd Intentional Conference on Advances in Strategic Technologies, August 12-14, 2003, Kuala Lumpur, Malaysia, pp: 683-688.

Priyank, K., 1997. VLSI logic test, validation and verification, properties and applications of binary decision diagrams. Lecture Notes, Department of Electrical and Computer Engineering University of Utah, Salt Lake City, UT., USA.

Raseen, M. and K. Thanushkodi, 2009. ROBDD-software and hardware representations. Int. J. Comput. Sci. Eng. Syst., 3: 115-120.

Rudell, R., 1993. Dynamic variable ordering for ordered binary decision diagrams. Proceedings of the International Conference on Computer Aided Design, November 7-11, 1993, Santa Clara, CA., USA., pp: 42-47.

Shelar, R.S. and S.S. Sapatnekar, 2001. Recursive bipartitioning of BDDs for performance driven synthesis of pass transistor logic circuits. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, November 3-9, 2001, San Jose, CA., USA., pp: 449-452.

Somenzi, F., 2001. Efficient manipulation of decision diagrams. Int. J. Software Tools Technol. Transfer, 3: 171-181.