Building Binary Decision Diagrams of large multipliers, 120 bits and beyond for formal analysis and verification

***Abstract*—Although Reduced Ordered Binary Decision Diagram (BDD) gives compact representations of many useful logic functions appearing in logic synthesis and verification, it explodes for multiplication circuits. In this paper, we construct BDDs for integer multipliers using variables in partial products of multiplications, which was originally proposed by Burch. We show by experiments that BDDs can be constructed from the netlists of multiplication circuits up to 120 bits by 120 bits multipliers, and formal verification of large various multipliers can be performed with the constructed BDDs as they are still canonical. Moreover, we give an algorithm that has been experimentally shown to be able to construct BDDs for the primary outputs of very large multipliers, such as 1,024 bits by 1,024 bits multipliers, directly from nothing. The precise sizes of such BDDs for large multipliers which grow in a cubic way with respect to the bit widths of the multipliers are shown. The generation algorithm is inferred from the shape of the BDDs constructed from the netlists for small multipliers, and its correctness has been confirmed by experiments. The BDDs for the primary outputs of large multipliers are useful to generate uniform test patterns for the on-sets, as it is easy to compute the exact size of the on-set for any sub-space of primary inputs. Those BDDs can be the base for logic optimizations for large multipliers including precise analysis of approximate circuits.**

***Index Terms*—Binary Decision Diagram, formal verification, integer multipliers, equivalence checking**

I. Introduction

Reduced Ordered Binary Decision Diagram (BDD in short) has been widely used in logic synthesis, verification, testing, and other fields in Electronic Design Automation (EDA). There have been a number of successes and now BDD is a part of essential tools in EDA technology. On the other hand, it is well known that BDD does not work well for complicated arithmetic computations, such as multiplication on integers, as its size grows exponentially with respect to the number of bits regardless of its variable ordering [3]. Multiplication circuits might not be necessarily synthesized or verified as they are already designed and kept in design libraries. People may not need to design new multipliers. Instead they can pick up appropriate circuits from the libraries. Now there are, however, fields that need customized and large multipliers. For example, for secured encryption, it is better to use large multipliers such as 256 bits or larger. Such multiplication circuits may not be found in the libraries, and a new design is developed for each application, as some sort of customization may be necessary for the application.

It may not be easy to develop such a large customized multiplication circuit as most of the formal methods available today, such as BDD, SAT (Satisfiability checking), exhaustive simulation, do not behave efficiently at all for multipliers. Experiments have shown that with SAT solvers, integer multipliers up to 12 bits can be analyzed, exhaustive simulation can practically work for up to 16 bits complete verification, and BDD can be built for up to 18 bits multipliers. As SAT solvers must perform almost exhaustive analysis for multipliers, it gives the worst performance among the three approaches. Actually BDD performs best among the three even though BDD size explodes for multipliers. In any case, all of these methods cannot be apparently applicable to large multipliers such as 128, 256 or large bits multipliers.

Binary Moment Diagram (BMD) [4] can work for large multipliers in terms of its size. The actual analysis used in formal verification and logic synthesis, however, needs word-level analysis as BMD is a word level representation of the logic, and it may not be straightforward to be applied to the analysis of real multiplier designs [5].

In this paper, we re-examine the use of BDD for the analysis of multiplication logic and its circuit implementations. Although BDD explodes for multipliers, there are extensions of BDD or extended use of BDD under which the sizes of the graphs do not explode exponentially. One common technique for such approaches is to allow the variables to appear more than once in the case spiting paths of the diagrams. In normal BDD, each variable can appear only once in any path from the root to the leaves. Then the size of normal BDD grows exponentially with respect to the bit width of the multipliers. The previous researches such as the ones proposed by J. Burch [1] and J. Jain et al. [6] have shown that the size of the diagrams used remain polynomial sizes for multipliers. Actually Index-BDD shown in [6] gives very small size diagrams for multipliers, but Index-BDD is not a canonical form and special analysis procedures must be applied to the formal analysis required in logic synthesis and verification. The method shown in [1] gives polynomial size (cubic or quadratic depending on the variable ordering) for multipliers and the diagram (actually it is normal BDD with different set of variables) remain canonical in the sense that with the same additional variables and ordering, the generated BDDs for the correct implementations of the multiplications become

the same.

The method shown in [1] may not be directly applicable to wide varieties of multiplication circuits, as it requires the use of partial products in the circuits. That is, the circuit must compute all partial products and then sum up all of them in some ways depending on the architectures of the multipliers. It cannot directly deal with multipliers which do not work that way, such as Booth multipliers. There are some discussions in [1] on how to deal with Booth multipliers, but it is out of the scope of this paper. Here we assume the multiplication circuits to be analyzed first generate all partial products and then they are summed up in some ways, which is one of the common circuit architecture for the integer multipliers used today.

Actually multipliers with various architectures can be generated automatically by automatic compilers, such as the one at [7] up to 64 bits and [8]. Different architectures of integer multipliers, such as array, Wallace tree, balanced delay tree, overturned-stairs tree, Dadda tree, (4:2) compressor tree, (7:3) counter tree, and redundant binary addition tree for partial product accumulator, and also ripple carry adder, carry look-ahead adder, Kogge-stone adder, Brent-Kung adder, Han-Carlson adder, carry select adder, conditional sum adder, ripple-block carry look-ahead adder, block carry look-ahead adder, carry-skip adder, Lander-Fischer adder as final stage adder, can be generated and be formally verified. The method shown in this paper can successfully deal with these varieties of multipliers.

The contributions of the paper are summarized as follows:

- It has been shown that BDD based formal analysis of large multipliers, 120 bits by 120 bits integer multiplications and beyond can be realized with the following two methods as shown in the experiments.
- Based on the idea of introducing additional variables introduced by [1], the BDDs for large multipliers up to 120 bits by 120 bits integer multiplications have been successfully constructed by traversing the netlists from inputs to outputs and are used for formal equivalence checking of multiplication circuits of various circuit architectures utilizing the fact that the generated BDDs are canonical with respect to the analysis of integer multiplications
- An algorithm by which the BDDs for the primary outputs of integer multiplications can be constructed directly from nothing, and it has been applied to the generation of BDDs for up to 1,024 bits by 1,024bits integer multiplication successfully. The BDDs for the primary outputs of large multipliers are useful to generate uniform test patterns for the on-sets, as it is easy to compute the exact size of the on-set for any sub-space of primary inputs. Those BDDs can be the base for logic optimizations for large multipliers as well.
- As the BDDs for large multipliers can be built, they may be used for various formal analysis including the precise analysis of approximate circuits. For example, under which input values the approximate circuits generates how much magnitude of errors can be precisely computed by analyzing the two BDDs for the correct and approximate circuits. Both of error magnitude and error frequency can be computed. This formal analysis becomes feasible for large multipliers, such as 120 bits by 120 bits multipliers and beyond.

This paper is organized as follows: The next section quickly reviews the approaches to the formal analysis of multipliers including decision diagrams, SAT, and algebraic methods. Then the discussions on BDD construction for integer multipliers with additional variables are given followed by the experimental results on up to 120 bits by 120 bits multipliers. In the following section, an algorithm to directly generate the BDDs for the primary outputs from nothing for very large multipliers, such as 1,024 bits by 1,024 bits multipliers is introduced. The final section gives concluding remarks.

II. Approaches to formal analysis and verification of multipliers

Here we would like to discuss the techniques by which formal equivalence checking of given two multipliers with different architectures can be performed. Typically they are based on the creation of the "miter" circuit of the given two multipliers, that is, the outputs of the two multipliers are connected to exclusive-OR gates which are further connected to an OR gate so that the output of the OR gate is always 0, if the two multipliers are logically equivalent. We can apply SAT solvers to see if the output of the OR gate can become 1. If SAT solvers return UNSAT, the two circuits are logically equivalent. When we construct BDD for the miter circuit, the resulting BDD should consist of only a constant 0 node, if the two circuits are equivalent. We can also exhaustively simulate the two circuits for all 2^{2n} input values to see the equivalence. We have performed experiments on these three approaches, and the results show the following: SAT based approaches can process up to 12 bits by 12 bits integer multipliers if the timeout is 1 day. The exhaustive simulation can process up to 16 bits by 16 bits multipliers with the same timeout. The BDD construction for the miter circuit scales up to 18 bits by 18 bits multipliers without node overflow which is $2^{31} - 1$ nodes during the BDD constructions.

These results may look like a little bit surprising, as SAT based approaches are widely believed to be better than BDD in general in terms of scalability. It may be explained in the following way. All of the three approaches basically need exponential order of operations when verifying the equivalence, as BDD explodes exponentially with the multiplication logic and various techniques which make SAT solvers efficient, such as implications, may not work well for the multiplication logic. If, regardless the approaches, we need exponential order of operations, how efficiently such exponentially many operations are performed may be a critical issue for the performance. In that sense, BDD utilize the most systematic way through incremental graph constructions whereas SAT solvers may not be efficient as they are based on case-spiting and backtracking. Exhaustive simulation is something in between.

Please note that in practice if the two multipliers to be compared can be structurally similar which may have many in-

ternal equivalent points. In such cases, the circuits can be partitioned into small sub-circuits and so the equivalence checking may be able to be performed after the partitioning. There are many techniques available for such cases which dramatically reduce the running time for the equivalence checking. One implementation for those techniques can be found as a "cec" command of the logic synthesis and verification tool, ABC [9]. Actually cec command works very well if the two multipliers to be compared are structurally similar or we can find a number of equivalent internal signals between the two circuits. Even the 256 bits by 256 bits multipliers can be quickly verified in those cases. The above experiments, however, used multipliers with different architectures and we have confirmed that they cannot be processed with cec command of ABC within 1 day.

The experiments suggest that BDD may not be so bad actually for multipliers with respect to the comparison to other approaches except algebraic methods. There are approaches based on algebraic treatment of logic circuits which have been shown to be very efficient on modulo multipliers, for example [10] and integer multipliers for example, [11]. Although they are very efficient for multipliers, they may not be good at normal control logic circuits. Also, the researches in that direction are still on-going and their application to logic synthesis and other applications are still to come. Therefore, in this paper, BDDs are used for the formal analysis and verification of integer multipliers. In order to avoid exponential explosion in terms of the number of the nodes in the BDDs for multipliers, additional variables are introduced when representing the BDDs, which was first introduced in [1] and is the base of the techniques discussed in this paper.

III. BDD CONSTRUCTION WITH ADDITIONAL VARIABLES FOR MULTIPLIERS AND ITS APPLICATION TO FORMAL VERIFICATION

In this section, we review the techniques for the BDD construction of multipliers with additional variables introduced by Burch in [1]. As well known, if BDDs are constructed for multiplication logic with primary inputs as the variables, the size grows exponentially with respect to the bit width of the multipliers [3]. If additional variables, however, are used, the size of such BDDs can remain polynomial (cubic or quadratic depending on variables orderings).

The BDD construction method works only for multipliers where first partial products are computed and then they are summed up in some ways. A most typical such multiplier is array multiplier, but there are varieties of ways to sum up the partial products: array, Wallace tree, balanced delay tree, overturned-stairs tree, Dadda tree, (4:2) compressor tree, (7:3) counter tree, and redundant binary addition tree for partial product accumulator, and also ripple carry adder, carry look-ahead adder, Kogge-stone adder, Brent-Kung adder, Han-Carlson adder, carry select adder, conditional sum adder, ripple-block carry look-ahead adder, block carry look-ahead adder, carry-skip adder, Lander-Fischer adder as final stage adder. Depending on these choices, various circuit architectures for integer multiplications have been proposed and used in real life. The method discussed in this paper can be applied to them. On the other hand, there are multipliers which do not work that way. One example is Booth multiplier where radix based computations are taken place first instead of simple partial products. In such circuits, the method discussed in this paper cannot be directly applied. Although there are some discussions on how to deal with those circuits in [1], it is out of the scope of this paper.

The way to introduce additional variables are as follows: For each occurrence of a primary input in the partial products computations, a different variable is used. For a N bits by N bits integer multiplication, there are in total N^2 different partial products and for each partial product, two independent variables are used. Therefore, in total there are $2N^2$ variables, and the BDDs are constructed with those variables instead of the original primary inputs.

Now would like to discuss the above additional variables for the BDD construction with small examples. Figure 1 shows a 4 bits by 4 bits array multiplier circuit assuming that all partial products, such as $x_0 \wedge y_0 (= x_0 y_0), , x_0 \wedge x_1 (= x_0 y_1), ..., x_3 \wedge y_3 (= x_3 y_3)$ are computed separately by another circuit. There are in total 8 primary inputs, $x_0, y_0, x_1, y_1, ..., x_3, y_3$. So the BDD constructed has these 8 variables. On the other hand, the BDD construction method in [1] uses $2 \times 4^2 = 32$ variables as shown in Figure 2.

Please note that each original primary input variable becomes four independent variables depending on how they are used in the partial products. For example, the primary input variable, x_0 becomes the following four variables, $x_0^0, x_0^1, x_0^2, x_0^3$, as can be seen in Figure 2. These four variables are essentially the same as the original primary input, x_0. But they are considered to be independent variables when constructing the BDD for the adder circuits shown in Figure 2. The interesting fact is that if we construct the BDD for the adder circuit with additional variables shown in Figure 2, the size remains polynomial with respect to the number of bits (cubic or quadratic depending on variable orderings). If we, however, add additional constraints that the four variable corresponding to the same primary input must have the same values (which is the case for the multiplication logic), the BDD size explodes exponentially. We can say that if the BDDs constructed with these additional variables for a multiplier known to be correct and the BDD for the target multiplier to be verified are homomorphic and the primary input variables are never used except for computing partial products, the target multiplier is proven to be correct.

Figure 3 and 4 show the case of a 4 bits by 4 bits Wallace multiplier. As primary inputs are used only for computing partial products in the case of Wallace multipliers, the additional variables are introduced in the same way as the case of array multipliers as can be seen from the figures. There have been proposed many multipliers of different architectures where the difference is how to sum up all the partial products. For such multipliers the additional variables can be introduced in the same way and the size of BDDs remain polynomial with those variables. The constructed BDDs remain canonical with the

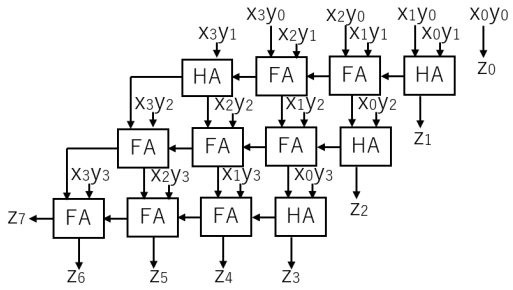


Fig. 1. 4 bits by 4 bits array multiplier

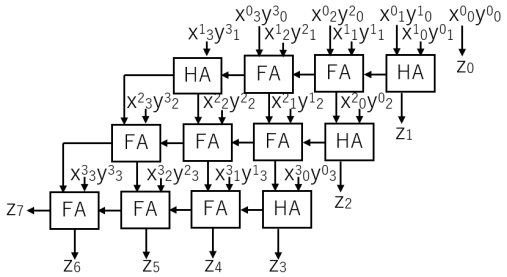


Fig. 2. Array multiplier with additional variables

additional variables.

In [1], two opposite variable orderings are mentioned: the high-to-low order and the low-to-high order. The high-to-low order for a 4 bits by 4 bits multiplier is shown in Figure 5 as red numbers. (1) is the first and (32) is the last. The low-to-high order is the opposite of it. The BDD size with respect to the number of bits for low-to-high orderings is suggested to be quadratic in [1] which has been confirmed experimentally up to 120 bits by 120 bits integer multipliers as shown by the experiments in the next section. Through our experiments we have come up with the exact 3rd order polynomial expression for high-to-low orderings. Moreover, as shown later, we have come up with an algorithm by which

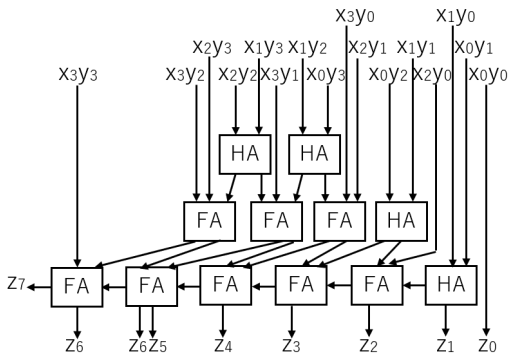


Fig. 3. 4 bits by 4 bits Wallace multiplier

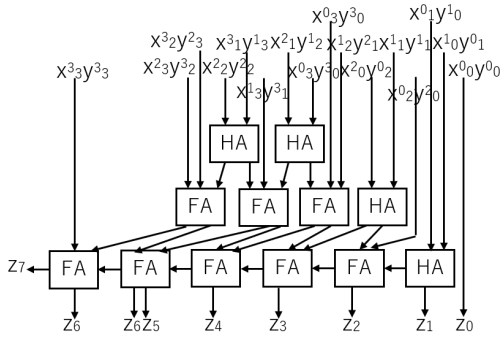


Fig. 4. Wallace multiplier with addtional variables

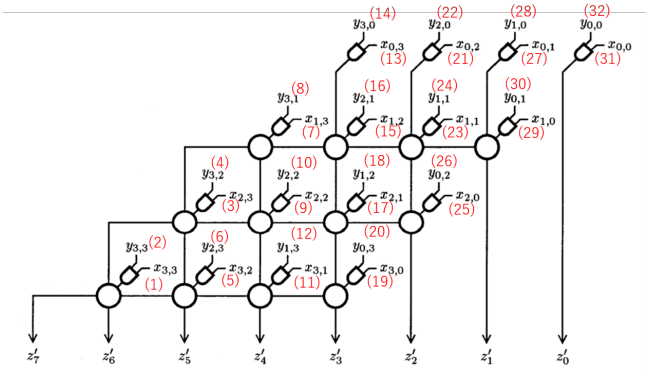


Fig. 5. High-to-low variable ordering is added to Figure 2 of [1]

BDDs for the primary outputs of any size multipliers. In fact we have successfully generated the BDD for 1,024 bits by 1,024 bits integer multipliers which has 2,144,335,879 nodes (the construction took around 66,100 seconds on Intel Xeon E5-2699 v4 @ 2.20GHz single thread).

Now we show our experimental results on the BDD construction with additional variables in the next section.

IV. Experimental results of BDD construction up to 120 bits by 120 bits multipliers

The experimental results on the BDD constructions with the regular BDD, the additional variable BDD with the low-to-high ordering, and the additional variable BDD with the high-to-low ordering are shown in Figure 6. The BDD package we use does not have dynamic reordering mechanism, such as the one shown in [12]. This is simply because the numbers of bytes to represent each node in BDD can be reduced without dynamic reordering, that is, no need of having pointers horizontally. As we know the variable orderings in advance, the same orderings can be used throughout the BDD constructions.

The machine we used is Intel Xeon E5-2699 v4 @ 2.20GHz single thread) having 512GB of memory. "NodeOver" means the number of BDD nodes exceeds the limit which is $2^{31} - 2$ nodes during constructing BDD by traversing the circuits from inputs to outputs. The BDDs we are constructing are the shared ones for all outputs with negative edges.

Size	AIG	Reguar BDD		L to H order		H to L order	
bits	nodes	Nodes	Time (sec)	Nodes	Time (sec)	Nodes	Time (sec)
16	1,872	41,421,674	318.7				
17	2,125	119,079,286	1,149.8				
18	2,394	343,517,163	5,304.1				
19	2,679	NodeOver					
...							
24	4,344			547,103	17.4	51,757	3.7
32	7,840			1,793,351	76.9	124,813	20.2
40	12,360			4,485,387	304.9	246,253	62.1
48	17,904			9,460,947	654.0	428,365	188.4
56	24,472			17,753,243	1,252.8	683,437	408.5
64	32,064			30,592,099	2,043.1	1,023,757	851.4
72	40,680			49,403,079	3,531.3	1,461,613	2,150.6
80	50,320			75,804,207	6,736.8	2,009,293	7,362.2
88	60,984			111,608,983	10,041.9	NodeOver	
96	72,672			158,827,519	15,577.8		
104	85,384			219,666,535	23,013.3		
112	99,120			296,529,359	34,719.0		
120	113,880			392,015,927	51,571.6		
128	129,664			NodeOver			

Fig. 6. BDD construction results with regular BDD, additional variable BDD with low-to-high ordering, and additional variable BDD with high to low ordering

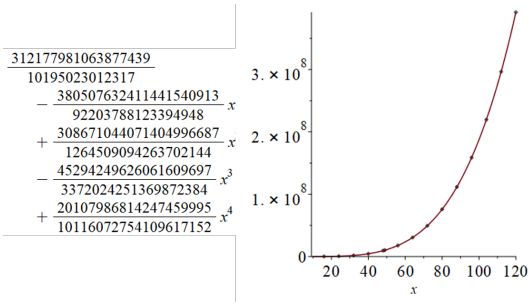


Fig. 7. BDD size with the low-to-high ordering

Here the multipliers are all integer array multipliers. As long as the multipliers have the circuit architectures which compute all partial products first and then sum them up, the results are quite similar, as they eventually generate the same BDDs. That is, regardless of the circuit architectures of the multipliers, equivalence checking and the formal verification is straightforward as long as their BDDs with additional variables can be constructed.

From the results, we can say the followings: With regular BDD (variables are primary inputs of the multipliers), BDDs for up to 18 bits by 18 bits multipliers can be built within the node number limit, and so clearly it does not scale. BDD constructions with additional variables have scaled much more. With the low-to-high ordering, up to 120 bits by 120 bits multiplier circuits have been processed successfully before NodeOver happens, and with the high-to-low ordering, up to 80 bits by 80 bits multiplier circuits have been processed successfully.

We think these results are amazing as it may be ways to utilize the presented approach not only in formal verification such as equivalence checking and model checking on circuits having large multipliers, but also in logic synthesis and optimization where efficient ways to represent and manipulate global don't cares are required for powerful methods, such as the ones shown in [13], [14]. Traditionally logic synthesis and optimization of circuits having complicated arithmetic units, such as multipliers are hard in terms of salability. The presented approach may open up new directions for the problems.

When we compare the BDD construction results with the low-to-high ordering and the high-to-low ordering, we can notice interesting facts. Firstly the BDD size with the low-to-high ordering increases in a quadratic way as mentioned in [1], although the precise numbers of nodes are slightly different as shown in Figure 7. In the figure x is the input bit width of the integer multipliers, and so if x is 64, the figure shows the number of nodes in the shared BDD of all outputs for the 64 bits by 64 bits multiplier The 4th order polynomial shown in the figure is the minimization result of least square errors. The polynomial is an approximation for the numbers of nodes in BDDs, but it show pretty good matching to the experimental results.

Secondly, the BDD size with the high-to-low ordering increases in a cubic way which is not mentioned in [1] as shown in Figure 8. The left part of the figure shows the matching between the experimental results and the polynomial, and the right part shows the polynomial up to the 300 bits by 300 bits multiplier. Interestingly, in this case, we can extract a precise 3rd order polynomial for the numbers of shared nodes of all outputs. It is $4x^3-6x^2-4x+13$ where x is the bit width of the multipliers, and this shows the exact numbers of the nodes, not an approximation. That is, the BDD size increases in a quadratic way with the low-to-high ordering whereas it increases in a cubic way with the high-to-low ordering. As can be seen from Figure 6, when we compare the corresponding BDD sizes for the same bit width multipliers, the high-to-low ordering gives much smaller BDDs, such as 1,023,757 nodes with the high-to-low ordering and 30,592,099 nodes with low-to-high ordering for the 64 bits by 64 bits multiplier. However, the BDD constructions with the high-to-low ordering fails for the multipliers whose bit width is larger than 80 whereas the BDD constructions with the low-to-high ordering succeed up to the 120 bits by 120 bits multiplier. This is because the BDD sizes in the intermediate steps of BDD construction by traversing the circuits from inputs to outputs grow more quickly with the high-to-low ordering than with the low-to-high ordering, although the final shared BDD sizes for all outputs are much smaller with the high-to-low ordering. In general we can observe that when constructing BDD from the circuits by traversing inputs to outputs, the sizes of BDDs for internal nodes grow up to some points and then reduce to the BDD sizes for primary outputs.

This phenomena happens more in the case of the high-to-low ordering than the case of the low-to-high ordering, which can be confirmed by observing the processing time for BDD constructions in Figure 6. The time for the BDD constructions is basically proportional to the total numbers of nodes of BDDs for internal and primary outputs of the multipliers. So it is

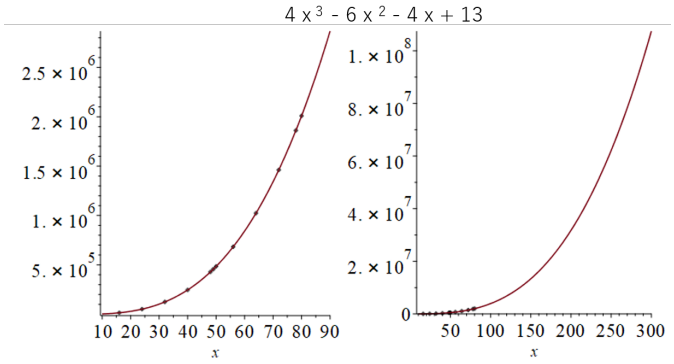
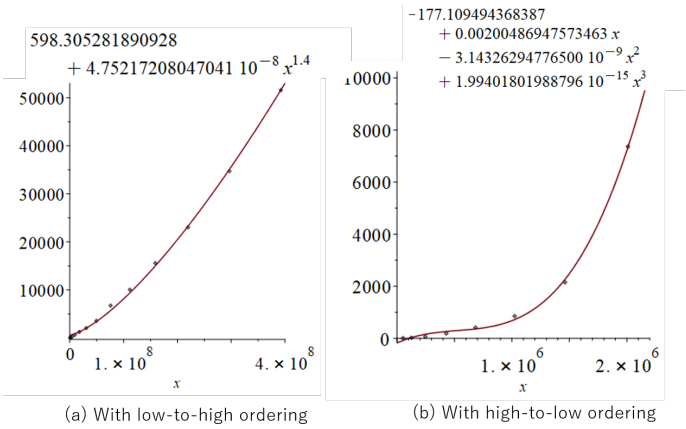


Fig. 8. BDD size with the high-to-low ordering



(a) With low-to-high ordering (b) With high-to-low ordering

Fig. 9. Relationship in the case of the low-to-high ordering between the numbers of nodes for the outputs of the multipliers and the processing time for the BDD constructions

proportional to the size of the multipliers (numbers of gates in the circuits). The numbers of AIG nodes in the multipliers extracted from Figure 6 are precisely $8x^2 - 11x$ where x is the bit width of the multiplier and grow in a square way as the target multipliers are integer array multipliers with ripple carry chains.

Figure 9 (a) shows the relationship in the case of the low-to-high ordering between the numbers of nodes for the outputs of the multipliers and the processing time for the BDD constructions. It is a little bit more than linear (proportional to $x^{1.4}$ where x is the bit width). Considering to the fact that the numbers of AIG nodes grow in a square way with respect to x, we may be able to conclude that the size of the BDDs for the intermediate nodes does not become much larger than the size of the BDDs for the primary outputs of the multipliers. That is the reason why the BDD constructions with the low-to-high ordering have been successfully up to the 120 bits by 120 bits multiplier.

On the other hand, Figure 9 (b) shows the relationship in the case of the high-to-low ordering between the numbers of nodes for the outputs of the multipliers and the processing time for the BDD constructions. The processing time grows in a cubic way with respect to the numbers of nodes in the BDDs for the primary outputs. As this is more than the growth of the numbers of AIG nodes which increases in a square way, the size of the BDDs with the high-to-low ordering for the internal signals of the multipliers increase more rapidly than the case of the BDDs with the low-to-high ordering, although the final size of the BDDs for the primary outputs of the multipliers with the high-to-low ordering is much smaller than that with the low-to-high ordering. This is an interesting phenomenon observed throughout our experiments. It may give us more insight into the BDD constructions for multipliers through future researches.

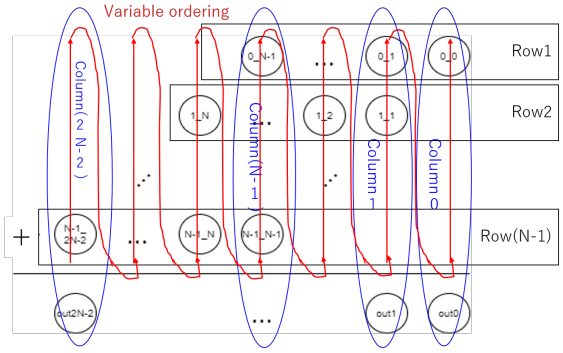


Fig. 10. Adder tree to be computed for the multiplication with the additional variables for the partial products

V. Direct Construction of BDDs for Primary Outputs of Very Large Multiplication

In this section, we would like to directly construct the BDDs for the primary outputs of the multipliers without building the BDDs for any other signals, such as the BDDs for the intermediate signals of the multipliers. We have analyzed the constructed BDDs with the high-to-low ordering generated through the experiments shown in Figure 6 and have found the following properties on the typologies of the generated BDD.

First of all, in order to compute the multiplication with the additional variables introduced in [1], the adder tree shown in Figure 10 is to be computed. Here our goal is to come up with the BDDs for this kind of adder trees with the high-to-low variable ordering which is shown as an red arrow in the figure. For each digit, the BDD is to be constructed without traversing the circuit which is a natural way to construct BDDs. Instead here it is constructed algorithmically.

The BDDs for the first four lower outputs of Figure 10 is shown in Figure 11. In the figure, the numbers of nodes for the variables are shown in blue. We can notice that these numbers are following some rules. For out_0 which is the first column, the logic is just AND of the two least significant bits and very simple. For out_1, out_2, and out_3, the bottom two nodes have the same topology, and the upper parts follow rules of the topology: For the first some numbers of variables, only one node exists and then the numbers of nodes are increased.

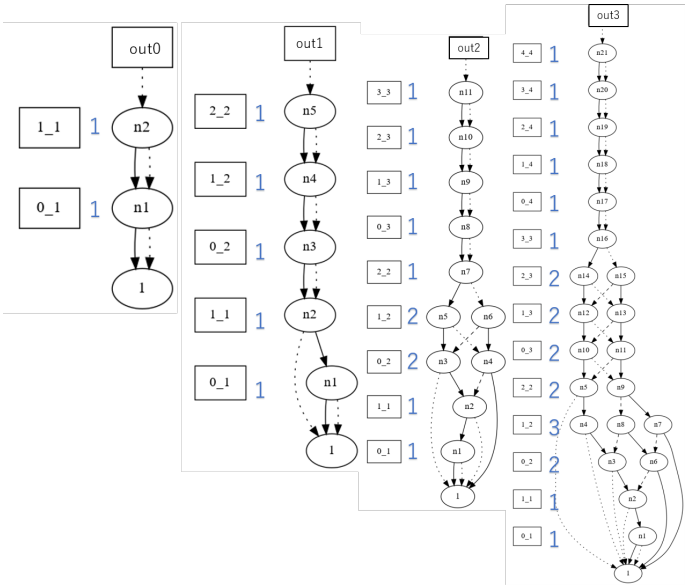


Fig. 11. BDD for each output of the adder tree in Figure 10

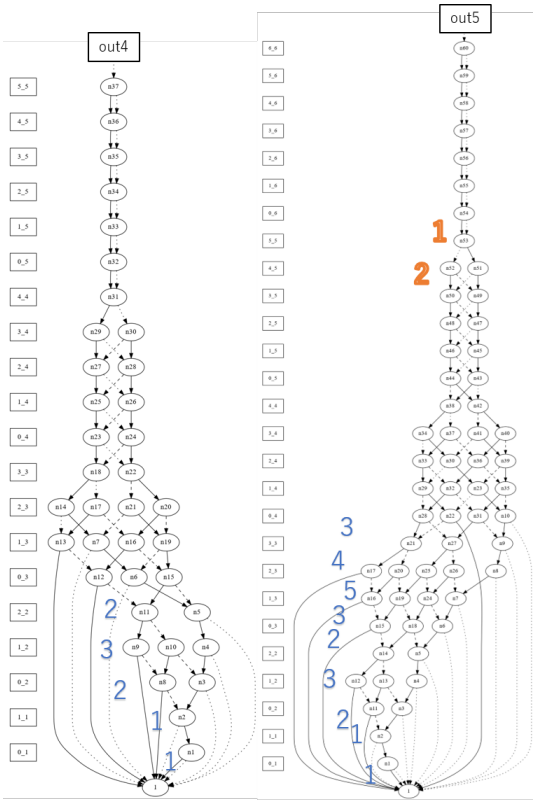


Fig. 12. BDD for larger outputs of the adder tree (1)

Figure 12 and Figure 13 show the BDDs for the 5th to 8th outputs of the adder tree. If we compare the numbers of nodes for the variables, when the output bit is increased by 1, the nodes are doubled. They are marked with the red numbers.

Also, as can be seen from the figures, the ways to connect among the nodes are very regular depending on the positions of the variables in the BDDs.

Based on these structural and inductive manual analysis, an algorithm by which the BDDs for the multipliers with the additional variables using the high-to-low variable ordering has been developed and is shown in Figure 14. We have estimated the numbers of the variables which have only one node, and the variables whose numbers of nodes are doubled, and use them to connects BDD nodes in the generation algorithm as shown in the figure. In the algorithm, "constant_1" means just 1 as a constant.

With this algorithm the BDD for the k-th output is constructed directly from lower output bits to higher output bits. When generating the BDD for the next output, by using unique table in the BDDs, the nodes in the new BDD is shared with the existing nodes in the previous BDDs if there is any. Therefore, when finishing the generation of the BDD for the largest output. the shared BDDs for all the outputs are obtained.

The direct BDD generation algorithm shown in Figure 14 has been implemented as a C program which has around 500 lines of codes. The experimental results are shown in Figure 15. As seen from the left part of the figure, with the restriction of the number of nodes in BDDs which is $2^{31}-1$, the BDD for all outputs of 1,024 bits by 1,024 bits integer multiplier has been successfully generated as shown in the figure. It took less than a day.

In the right of the figure, a graph between the numbers of the nodes in the BDDs and the time for the BDD generation is shown. As seen from the graph, the generation time roughly increases in a square way with respect to the numbers of the nodes in the BDDs. This may be due to the generation algorithm. It constructs BDDs for the outputs one by one instead of generating the entire BDD for all the outputs at once. There can be some redundant processes in our algorithm.

The numbers of nodes shown in the experimental results precisely follow the experimentally inferred numbers of nodes, $4x^3-6x^2-4x+13$ where x is the bit width of the multipliers. If necesary, we can construct BDDs for larger multipliers by allowing larger numbers of nodes in the BDDs when implementing the algorithm.

VI. Conclusions

In this paper, we have constructed BDDs for integer multipliers using variables in partial products of multiplications, which was originally proposed in [1]. We have shown by experiments that BDDs can be constructed from netlists of multiplication circuits up to 120 bits by 120 bits multipliers, and formal verification of large various multipliers can be performed with the constructed BDDs as they are still canonical. Moreover, based on the structural and inductive manual analysis, we have come up with an algorithm that has been experimentally shown to be able to construct BDDs for the primary outputs of very large multipliers, such as 1,024 bits by 1,024 bits multipliers, directly from nothing. The precise sizes of such BDDs for large multipliers which grow in a

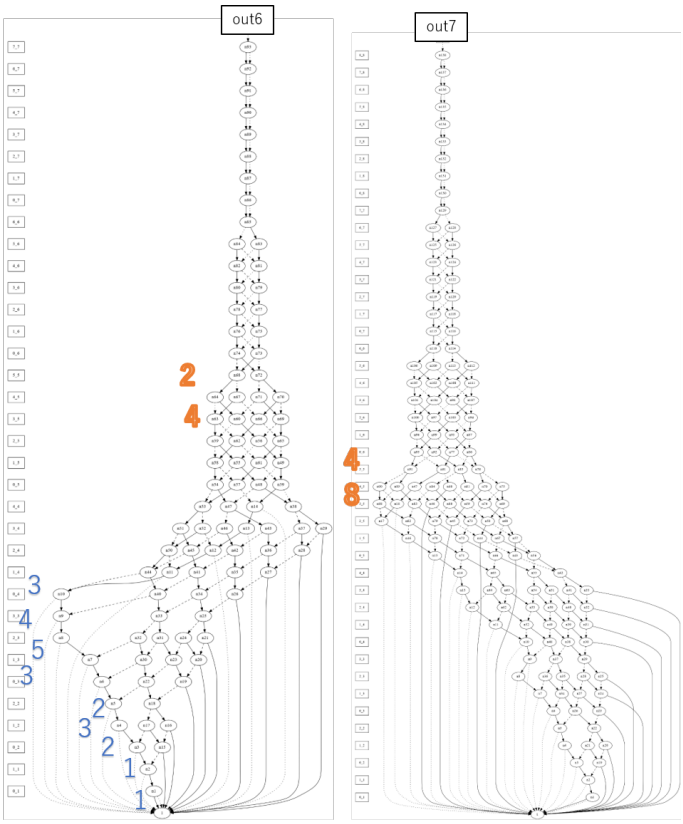


Fig. 13. BDD for larger outputs of the adder tree (2)

- Let list **y** have only constant_1
- for(int column=1; column < K; column++) // column 0 is easy
  - Clear list **x**
  - length = min(2^(K-column), column+row);
  - if(length > length(**y**) // In this case, length(**y**) should be lenght-1
    - Insert constant_1 to **y** as the next-to-top element
  - Create a node having **y**'s top element as positive cofactor and y's 1%length element as negative cofactor and it as **x**'s top element
  - for(int i = 1; i < length; i++)
    - Create a node having **y**'s i-th element as positive cofactor and y's (i+1)%length element as negative cofactor and it as **x**'s i-th element
    - Swap **x** and **y**
  - length = min(2^(K-column-1), column);
  - if(2*length > length(**y**) { // In this case, length(**y**) should be 2*lenght-1
    - Insert constant_1 to **y** as the next-to-top element
  - Create a node having **y**'s top element as positive cofactor and **y**'s 1%length element as negative cofactor and it as **x**'s top element
  - for(int i = 1; i < length; i++)
    - Create a node having **y**'s 2*i-th element as positive cofactor and **y**'s (2*+1)-tb element as negative cofactor and it as **x**'s i-th element
- for(int row = 0; row <= K; row++)
  - Create a node having **y**'s top element as positive cofactor and complement of **y**'s top element as negative cofactor and make it **y**'s top element
- Complement of **y**'s top element is the root of the target BDD

Fig. 14. Direct BDD constructions for the primary outputs of the multipliers with the high-to-low ordering

Bit width	Nodes	Time (sec)
32	124,813	0.03
64	1,023,757	0.33
128	8,289,805	4.27
256	66,714,637	189.79
384	225,606,157	1,076.61
512	535,296,013	3,487.90
640	1,046,115,853	8,706.60
768	1,808,397,325	18,144.60
896	2,872,472,077	34,621.72
1,024	4,288,671,757	66,109.17

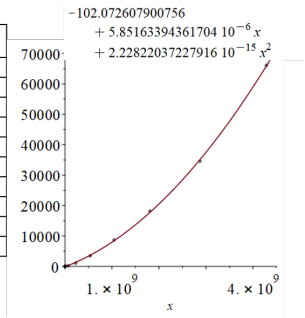


Fig. 15. Direct BDD constructions for the primary outputs of the multipliers with the high-to-low ordering

cubic way: $4x^3-6x^2-4x+13$ where x is the bit width of the multipliers, with respect to the bit widths of the multipliers are shown. The BDDs for the primary outputs of large multipliers are useful to generate uniform test patterns for the on-sets, as it is easy to compute the exact size of the on-set for any sub-space of primary inputs. Those BDDs can be the base for logic optimizations for large multipliers as well. For example. in the existing synthesis methods for approximate multipliers, only the largest error magnitude or error frequency are analyzed and guaranteed. With the BDDs which can be constructed with the present methods, an exact analysis of error magnitude and error frequency can be given, which may improve the usefulness of the approximation circuits.

REFERENCES

[1] Jerry R. Burch: Using BDDs to Verify Multipliers, *DAC*, 1991.
[2] Randal E. Bryant: Symbolic manipulation of Boolean functions using a graphical representation, *DAC* 1985, pp. 688-694.
[3] R.E. Bryant: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication, *IEEE Transactions on Computers*, Vol.40, No.2 (February, 1991), pp. 205–213.
[4] Randal E. Bryant, Yirng-An Chen: Verification of Arithmetic Circuits with Binary Moment Diagrams, *DAC*, 1995, pp. 535-541.
[5] K.Hamaguchi, A.Morita, S.Yajima: Efficient construction o fbinary moment diagrams for verifying arithmetic circuits, *International Conference on Computer-Aided Design*, November, 1995.
[6] Jawahar Jain, James R. Bitner, Magdy S. Abadir, Jacob A. Abraham, Donald S. Fussell: Indexed BDDs: Algorithmic Advances in Techniques to Represent and Verify Boolean Functions, *IEEE Trans. Computers*, 46(11), 1997, pp. 1230-1245.
[7] https://www.ecsis.riec.tohoku.ac.jp/topics/amg/i-amg/
[8] http://www.sca-verification.org/
[9] Robert K. Brayton, Alan Mishchenko: ABC: An Academic Industrial-Strength Verification Tool, *22nd International Conference on Computer Aided Verification (CAV 2010)*, 2010, pp. 24-40.
[10] Jinpeng Lv, Priyank Kalla, Florian Enescu: Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits, *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(9), 2013, pp. 1409-1420.
[11] Daniela Ritirc, Armin Biere, Manuel Kauers: Column-wise verification of multipliers using computer algebra, *FMCAD*, 2017, pp. 23-30.
[12] Yukio Miyasaka, Alan Mishchenko, Masahiro Fujita: A Simple BDD Package without Variable Reordering and Its Application to Logic Optimization with Permissible Functions. *International Workshop on Logic and Synthesis (IWLS)*, June, 2019.
[13] Subarnarekha Sinha, Robert K. Brayton: Implementation and use of SPFDs in optimizing Boolean networks, *ICCAD*, 1998, pp. 103-110.
[14] Yusuke Matsunaga, Masahiro Fujita: Multi-level logic optimization using binary decision diagrams, *ICCAD*, 1989, pp. 556-559.