Logic synthesis and verification on fixed topology

Masahiro Fujita
University of Tokyo

Alan Mishchenko
University of California, Berkeley

***Abstract*—We discuss about logic synthesis and formal verification of combinational circuits mapped to a given fixed topology. Here "fixed topology" means that circuit structures in terms of net lists except for gate/cell types are fixed in advance. That is, for logic synthesis, what should be generated are the types of gates/cells (or simply gates) in the circuits, and all the others are prefixed before synthesis. As the circuit topology is fixed in advance, placement and routing could be shared among different designs and minimum ECO can be realized by keeping the same layout. Also, we can show that we do not need many test vectors in order to guarantee 100% correctness of such synthesis. Small numbers of test vectors, such as only 100 test vectors for 30 input circuits, are sufficient to test if the circuits behave correctly for all possible input value combinations. That is, very efficient formal verification can be realized through simulations with small numbers of test vectors. We present SAT based implementation of the synthesis and a test vector generation method with preliminary but encouraging experimental results.**

I. Introduction

Actual circuit performance largely depends on its layout, i.e., where each gate is placed and how they are interconnected on a chip. If some circuit topology changes due to some reasons including changes of specification and design debugging, which are called as logic Engineering Change Order (ECO), more efforts may have to be spent for the layout of new designs. The problem is that even if the changes of performance is within small area or within small numbers of modules, surrounding modules may have to accommodate them in order for the entire design to be able to operate correctly with acceptable performance. It is preferable to keep the same topology of circuits even after various ECO happens, if ever possible.

There have been conducted various researches on logic synthesis for ECO. Recent ones include [1], [2], [3]. All of them try to minimize the changes in the resulting circuits after logic ECO. In this paper, we aim to do an extreme, that is, keeping the topology of the circuits after logic ECO exactly the same as the one before ECO.

Now we present an example in order to clarify the discussion. Figure 1 shows a typical logic design of full adder by using two half adders as shown in the figure. If we assume that the technology library has only various AND-INVERTER combined cells, the design in Figure 1 can be implemented as shown in Figure 2, which has nine gates. Now assume that this circuit has been placed and routed. Then assume that we like to realize subtraction based on 2's complement representation of numbers instead of addition. Instead of synthesizing a new circuit for the subtraction, we like to generate a circuit which has exactly the same topology of the circuit shown in Figure 2. That is, here the goal is to implement the subtraction on the circuit shown in Figure 3 where boxes are some types of gates which are to be determined when some circuits are mapped on to this circuit. As well known, subtraction based on 2's complement representation of numbers can be realized with an adder by complementing the number which is subtracted from the other number and assigning 1 to the carry input. The circuit of this implementation is shown in Figure 4. Please note that the three gates connected to the input, B, are complemented with respect to the circuit shown in Figure 2. Also the carry input, C_i, is assigned 1.

978-1-4799-6016-3/14/$31.00 © 2014 IEEE

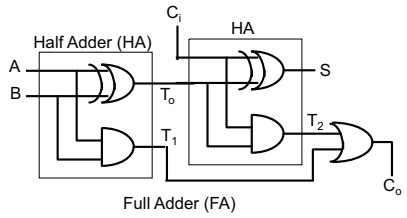


Fig. 1. An implementation of full adder

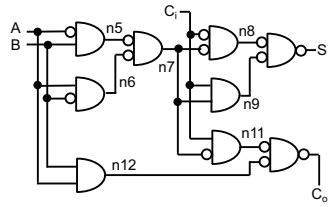


Fig. 2. Two-input AND-Inverter implementation of full adder

In this paper, we deal with combinational circuits or bounded time frame expanded sequential circuits. The goal of the logic synthesis in this paper is the following: Given the skeleton circuit where the circuit topology is fixed but types of gates are not fixed, and also given new design/specification, determines the types of gates in the skeleton circuit so that the resulting circuit becomes logically equivalent to the given design/specification. We can use exactly the same placement and routing on a chip as the topology of the circuit remains the same, and the performance of the circuit is very predictable.

In this paper, we also discuss about how to formally verify the circuits mapped to a fixed topology. If there are n inputs in a given combinational circuit, in order to formally (or completely) verify the circuit, we need to analyze or simulate with all possible input value combinations which are 2^n cases. Under fixed topology, however, the numbers of test vectors

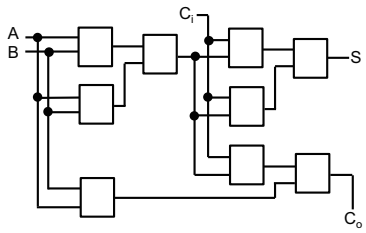


Fig. 3. Circuit in Figure 2 without gate types

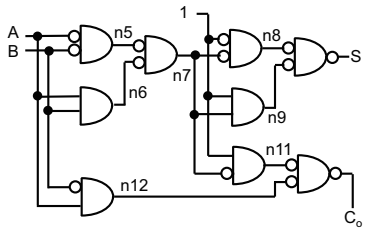


Fig. 4. A subtractor implementation

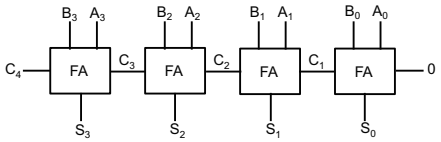


Fig. 5. Four bit adder based on full adders

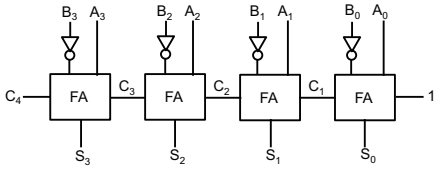


Fig. 6. Four bit subtractor based on four bit adder

required for "complete" verification can be very small. Actual experiments show even for 20-30 input circuits, we need only one hundred or so test vectors rather than $2^{20} - 2^{30}$ ones. This means that if the circuit mapped to a fixed topology behaves correctly for those hundreds input vectors, mathematically there is no way for the circuit to behave incorrectly for the other input vectors. We present a method by which such complete test vectors can be generated from the given topology and design/specification.

The proposed logic synthesis and formal verification can be naturally used for FPGA circuits where only LUT (Look Up Table) part is reprogrammed for a new circuit with no changes in routing among LUTs. Under this situation, the delays of the FPGA circuits basically do not change even after reprogramming.

The rest of the paper is organized as follows. In the next section, we discuss illustrative examples for our logic synthesis as well as formal verification. Then we show the proposed logic synthesis and formal verification method and its simplified one followed by their preliminary experimental results. The last section gives concluding remarks.

II. Illustrative examples

In this paper, we present methods for logic synthesis and formal verification for circuits mapped to a given fixed topology. As shown in the previous section, adders and subtractors can be mapped onto the same topology. In general sometimes different designs can be realized on the same topology circuits, but in many cases, they cannot. When successful, however, same layout can be used for those designs, and as a result, their performance is very predictable. Our proposed methods can quickly determine whether a given design/specification can be mapped to the given fixed topology or not. Moreover, they also generate a complete set of test vectors by which logical correctness of the mapped circuits can be checked assuming that the connections among gates are guaranteed not to change, i.e., possible logical bugs and manufacturing faults are only on types of gates. These sets of test vectors are very useful not only for logical correctness checking but also for testing manufacturing faults, as every fault inside the gates is covered.

In this section, we continue the discussion on adders/subtractors in the previous section in order to illustrate these claims. As discussed above, subtraction with 2's complement representation can be realized with an adder by complementing the inputs whose value is subtracted from the other input value and assigning 1 to the carry input. This is true for any bit-width subtraction. Figure 5 shows an implementation of 4 bit adder with full adders and ripple carry chains. In order to change this circuit to perform subtraction, i.e., $A - B$, B input is complemented and the carry input is given the value 1 as shown in Figure 6.

Here we use the same circuit for all of the four full adders in the four bit adder, whose AND-INVERTER gate implementation is shown in Figure 2. The subtractor design shown in Figure 6 can be implemented by replacing each of the full adders in Figure 5 with the subtractor shown in Figure 4. Please note that with the use of subtractor shown in Figure 4 in the locations of the full adder in Figure 5, it becomes the circuits shown in Figure 6. Also, the full adder shown in Figure 2 and the subtractor in Figure 4 have the same topology which is shown in Figure 3.

In order to represent all possible designs which can be mapped to the given circuit topology, we use LUT (Look Up Table) for each gate in the circuit. As LUT can represent all logic functions with the set of inputs, by appropriately programming the set of LUTs, all possible designs with that topology can be represented. So the problem to be solved is the following: Given circuit topology, *topology*, and given design, *design*, check if *design* can be implemented with *topology* by appropriately programming LUTs.

Our proposed logic synthesis and formal verification methods are looking for counter examples which make *topology* with LUTs behave differently from *design*. The counter examples generated become the set of test vectors for formal verification of the implementation on the fixed topology, *topology*. This is achieved by repeatedly solving SAT problems whose

solutions are counter examples, i.e, new test vectors, which cover the uncovered input space by the set of existing test vectors. If the SAT problem becomes unsatisfiable, there is no more counter example. That is, it means that the set of the generated test vectors cover all possible cases and the given design is correctly and successfully mapped to the given topology. If there is no restriction in circuit topology, the number of required test vectors is 2^n where n is the number of input in the design. If we map the design into a fixed topology, however, the numbers of required test vectors are very small, for example, we need only hundreds of test vectors for 20-30 input designs. Although there is no theoretical proof for this, experimentally there have been the case.

Now Let us see how many test vectors we need for the cases of the four bit adder and the four bit subtractor. Our proposed methods can determine the number of test vectors to check if four bit adder can be implemented with the circuit topology shown in Figure 5 where each full adder block consists of the circuit topology shown in Figure 3. As there are 9 two-input AND-INVERTER gates in the full adder, there are totally 36 two-input gates in the circuit topology for four bit adder. We represent each two-input gatel with a two-input LUT. If we can find an appropriate logic function for each LUT by which the entire circuit becomes logically equivalent to four bit adder, the mapping process is successful. We check this by repeatedly generating counter examples as test vectors for the equivalence checking until we cover all primary input space. If we apply the proposed method to this four bit adder, we need only 19 test vectors in order to "implicitly" cover all of the primary input space which are $2^9 = 512$ value combinations as there are nine primary inputs. On the other hand, if we map the four bit subtractor to the same circuit topology as four bit adder, this time we need 22 test vectors to cover all primary input space. These set of test vectors guarantee the correctness of mapping completely, that is, simulating with that set of test vectors is essentially equivalent to formal verification.

Now we show that a slight optimization for the four bit adder makes the circuit incompatible to four bit subtractor, that is, the same topology of circuit with this optimization cannot accommodate both of four bit adder and four bit subtractor. As we do not need to have carry-in input for the whole four bit adder, which means the carry-in input always get the value 0, the full adder for the least bit can be simplified to the one shown in Figure 7. Similar simplification is possible for the case of four bit subtractor where the full adder for the least bit can be simplified to the one shown in Figure 8 as it always get the value 1 for the carry input. By comparing the two circuits shown in Figure 7 and Figure 8, we can easily conclude that the circuit in Figure 8 can never be realized in the circuit in Figure 7 only by modifying the types of gates. Our proposed logic synthesis and formal verification method can quickly recognize this. Please note that, the mapping with the other direction is feasible. That is, four bit adder with the optimization can be mapped to the four bit subtractor with the optimization.

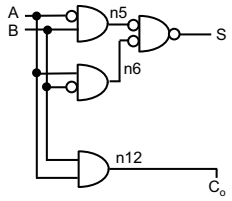


Fig. 7. An optimized circuit for the least significant bit of the four bit adder

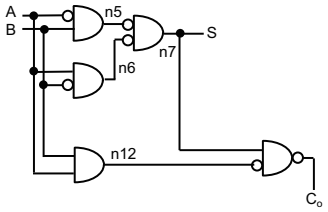


Fig. 8. An optimized circuit for the least significant bit of the four bit subtractor

III. Proposed synthesis and formal verification methods

In our method we first replace all of the gates in the given circuit topology with LUTs. For easiness of presentation, we assume all gates are two-input ones in this paper, but it is clear that the methods discussed can be extended with more than 2 inputs. As shown in Figure 9, we need four variable, $x_,x_2, x_3, x_4$, in order to represent the values of four rows in the truth table. The two inputs, in_1, in_2 control the multiplexers. Depending on the values of the inputs, right variable is selected out of x_1, x_2, x_3, x_4 in order to give the right value in the truth table.

We assume in this paper that the circuit topology is given in terms of two-input gates. If there are m gates in the given topology, there are a set of m of x_1, x_2, x_3, x_4 variables in the circuit with LUTs. The goal here is to find appropriate values for those $4m$ variables which make the circuit logically equivalent to the given design or specification. We try to solve this problem by repeatedly finding a solution candidate for the programming of LUTs and its counter example, if it is not actually a real solution. If we reach the situation where there is no counter example, we can conclude that the current solution is a real one.

Now let us define the problem formally and present our proposed method. Let in and x be sets of primary inputs and variables for LUTs respectively. Also let $SpeC(in)$ represent the correct values of outputs of the given designs/specifications and $Circuit(in, x)$ represent the values of the outputs of the fixed topology circuit with LUTs. Then the goal is to find the appropriate values for x which makes $Circuit(in, x)$ always equivalent to $SpeC(in)$. That is, it is to solve the following problem:

$$\exists x. \forall in. Circuit(in, x) = SpeC(in) \qquad ①$$

This is a QBF (Quantified Boolean Formula) and its satisfiability problem belongs to P-Space complete. In general QBF satisfiability can be solved by repeatedly applying SAT

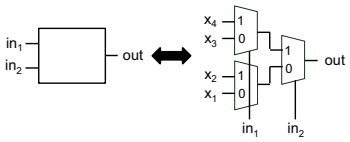


Fig. 9. Two-input LUT with four variables as values of truth table

solvers, which was first discussed under FPGA synthesis in [6] and in program synthesis in [7]. The techniques shown in [8] give a general framework on how to deal with QBF only with SAT solvers. These ideas have been applied to so called partial logic synthesis in [10]. Although our formulation of the problem is the same as the techniques shown in [10], our application is different in that we introduce LUTs to all gates rather than some subsets. Moreover, as shown later, our implementation is purely based on a single SAT solving with dynamic additions of constraints. That is, starting with a satisfiable problem, each time we find a test vector, we add additional constraints dynamically until the overall problem becomes unsatisfiable.

The basic idea is to eliminate non-solutions incrementally until there only remain real solutions. Here non-solution means that under that programming of LUTs the circuit is not equivalent to specification, and there has been found a counter example for that non-equivalence. So, the SAT problem to be solved first is the following:

$$\exists in, x.Circuit(in, x) \neq Spec(in) \qquad ②$$

Let the solutions for ② be in_1 and x_1. Under the programming corresponding to the value x_1, the circuit behaves differently from the specification with the input value in_1. Now we should look for solutions for programming of LUTs which behave correctly under input in_1. This constraint for programming of LUTs can be described as follows:

$$\exists x.Circuit(in_1, x) = Spec(in_1) \qquad ③$$

Please note that in_1 is a constant value and so the right hand side of the equation is just constant.

The conjunction of ① and ③ is the next SAT problem to be solved:

$$\exists in, x.\forall in.Circuit(in, x) = SpeC(in) \\ \wedge Circuit(in_1, x) = Spec(in_1) \qquad ④$$

Let the solutions for ④ be in_2 and x_2. Under the programming corresponding to the value x_2, the circuit behaves differently from the specification with the input value in_2. Now we should look for solutions for programming of LUTs which behave correctly under input in_1 as well as in_2, which can be obtained by solving the following SAT problem:

$$\exists in, x.Circuit(in, x) = SpeC(in) \\ \wedge Circuit(in_1, x) = Spec(in_1) \\ \wedge Circuit(in_2, x) = Spec(in_2) \qquad ⑤$$

We keep doing this until the SAT problem becomes unsatisfiable. If the number of iterations is p, the set of $in_1, in_2, ..., in_p$ is a complete set of test vectors by which we can determine if the circuit is correctly implementing the new design/specification. This is because of the fact that the SAT problem is unsatisfiable guarantees there is no more counter example. That is, if the circuit behaves correctly under $in_1, in_2, ..., in_p$, the circuit is guaranteed to behave correctly under all possible primary input value combinations.

Please note that the SAT problems have more and more constraints and no constraints deleted in the later iterations. This means that the learnings/implications learned when solving the previous SAT problems are guaranteed to be valid for later SAT problems. That is, all conflicts (or backtracks) made in the previous SAT solving processes are also to be made in the later SAT solving processes. So by transferring learnings/implications to the later SAT solving processes, we can essentially resume searching for solutions at the location where the previous SAT solving process has found a solution instead of starting the SAT solving process from the beginning. This significantly speeds up the entire SAT solving process.

Assuming that we are using case-splitting based SAT solvers, which are common nowadays, the discussions can be summarized as follows:

1) Start the SAT solver with the constraint ①
2) Continue case-splitting until a solution is found. If case-splitting finishes, i.e., case-splitting covers entire search space, go to 5).
3) Create a new constraint based on the solutions for inputs and LUT variables, e.g., ②
4) Generate a new SAT problem having the new constraint and go back to 2)
5) The set of solutions obtained is a complete set of test vectors by which we can make sure if $Spec(in)$ can be implemented on the topology of $Circuit(in_1, x)$. Therefore, solve the following SAT problem. If that is unsatisfiable, there is no way to map the new design/specification to the topology of the original circuit. If there is a solution, that is a way to map the circuit.

$$\exists x.Circuit(in_1, x) = SpeC(in_1) \\ \wedge ... \wedge Circuit(in_n, x) = Spec(in_n) \qquad ⑥$$

In the step 2) above, we continue or resume the case-splitting processes instead of restart so that all the previous reasoning can be skipped. As can be seen from the above, it is solving an unsatisfiable SAT problem by starting a satisfiable problem and adding new constraints each time a solution, i.e., test vector, is found until the entire formula becomes unsatisfiable. Therefore, the complexity of the proposed method is somehow close to the one for equivalence checking based on SAT where we have to make sure that there is no counter example by implicitly enumerating all solution space.

Now we show example traces of our proposed logic synthesis and formal verification method applied to the four bit adder shown before. The target circuit topology has 9 primary inputs, 5 primary outputs, and 36 two-input gates. The first SAT problem corresponding to ② has 519 clauses. After 8 conflicts/backtracks, a SAT solver finds a solution.

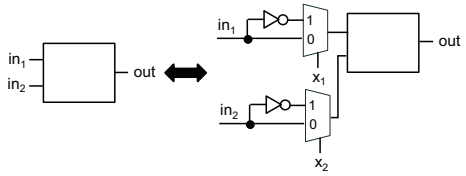


Fig. 10. Gate whose input polarities can be changed

This becomes the first test vector. Then add the constraints having 562 clauses which correspond to ④ as well as learnings/implications learned through the 8 conflicts/backtracks. With this new additional constraint, the SAT solver needs 2 more conflicts/backtracks to find the second solution, which becomes the second test vector. The new constraint from this second solution as well as newly learned learnings/implications has 1643 clauses. The SAT solver needs 20 additional conflicts/backtracks to find the third solution. This process continues for 19 iterations in total, and there are 19 test vector generated. After these, the SAT solver keeps searching for solutions, but there is no solution and the SAT solver reports the problem becomes unsatisfiable. The final constraint has 11,197 clauses and needs 8,014 conflicts/backtracks to show the unsatisfiability.

In order to process larger circuits, we would like to introduce a simplified model where each gate in the circuit can modify only its input polarities rather than arbitrary changes in its logic function when trying to accommodate a new design/specification.

A. A simplified method: Changes of input polarities in each gate for new designs/specification

The method disussed so far is a sort of most general method in the sense that if there is a way to map the given design/specification to the target topology, it surely finds one as long as the method can finish its computations. As can be seen from the experimental results in the following section, for large circuits, however, this method cannot finish its computations. In order to deal with larger circuits, there should be some simplification or restrictions in the selection of logic functions to be realized with each gate. There are varieties of ways for such restrictions, and there are obviously trade-offs in terms of in-feasibility of implementation and how large circuits can be processed. If we take a look at the adders discussed above, we can recognize that subtractors can be implemented on the topology of adders by only changing some of the input polarity of gates. Therefore, here we propose a simplified method where only input polarities of gates can be changed in order to accommodate new designs/specifications.

In the proposed simplified method, instead of using LUT, each gate is replaced with the circuit shown in Figure 10. In this circuit, by changing the values of controls for the two multiplexers, the polarities of inputs can be changed.

IV. Experimental results

The proposed methods have been implemented on top of the tool ABC [11] including the use of previously learned clauses in later SAT solving. A given circuit is first transformed into one only with two-input AND gates which may have inverters in its inputs and outputs if necessary. This is a sort of AND-INVERTER Graph (AIG) representation, and each gate has exactly two inputs. This gives us the topology for the circuit. Then the problem to be solved here is to check if the given original circuit which may have arbitrary gates can be mapped to this circuit topology. As the topology is generated from the original circuit, obviously this is always the case. In the experiments, our proposed methods confirm this and during the processes, they also generate complete sets of test vectors by which the correctness of mapping can be 100% checked.

The experimental results on ISCAS89 benchmark circuits are shown in Table I and Table II. Although ISCAS89 circuits are sequential ones, we only deal with their combinational parts only. So the numbers of inputs are the sum of primary inputs and flipflops, and the numbers of outputs are the sum of primary outputs and flipflops.

Name is the name of an ISCAS89 benchmark circuit, and PI/PO/FF/AND are the numbers of primary inputs, outputs, flipflops, and AND gates with possible inverters in their inputs and outputs. Vars/Clauses/Conflicts are the numbers of SAT variables, clauses, and conflicts, and Tests is the total number of test vectors generated using the proposed methods. Time is the processing time on a server computer having Linux kernel 2.6.32 64-bit, Dual Xeon E5-2690 2.9GHz, 128GB memory. ”>100h” means the computation does not finish in 100 hours and intermediate results are shown in other columns.

Form the Table I, we can observe the followings for the LUT based method:

- Circuits having up to around 30 inputs and 300 gates can be mostly processed whereas any larger circuits mostly cannot
- The numbers of complete test vectors are around hundreds even for circuits having 50 or more inputs

Form the Table II, we can observe the followings for the polarity based method:

- All ISCAS89 circuits can be processed
- The numbers of complete test vectors are around hundreds even for the largest circuits having more than ten thousands gates and one thousand inputs

As polarity based method targets subsets of the LUT-based method and it is much quicker to process, Table III shows the results by first applying the polarity method followed by the LUT based method. First a set of test vectors are generated with the polarity method and they are used as the initial set of test vectors for the LUT-based method. As can be seen from the table, comparing with the cases where the LUT based method is applied, numbers of test patterns remain similar but processing time is up to 8 times faster.

The last experiment is to map 12-input logic functions to a cicuit which is a collection of LUT as show in Figure 11. The goal here is to synthesize the configuration of the LUTS in the circuit from given specifications. There is a research on enumerating all logic functions appearing in various

Name	PI	PO	FF	AND	Vars	Clauses	Conflicts	Tests	Time (s)
s27	4	1	3	8	957	1849	553	16	0.05
s298	3	6	14	102	61303	147296	160601	101	13.08
s344	9	11	15	105	60600	149232	207211	90	13.76
s349	9	11	15	109	54532	134800	194326	78	11.28
s382	3	6	21	140	78768	188852	89182	96	8.43
s386	7	7	6	166	223613	548144	1226183	243	216.57
s400	3	6	21	148	74175	179106	94104	85	8.47
s444	3	6	21	155	135334	333328	148896	145	35.21
s510	19	7	6	213	108051	276144	5140718 10	83	>100h
s526	3	6	21	203	228902	565723	1535242	192	432.84
s641	35	24	19	146	230163	551977	28722305	242	11739.97
s713	35	23	19	160	285027	699826	7530138	272	1959.96
s820	18	19	5	345	304620	763856	383490557	154	>100h
s832	18	19	5	356	348639	874715	282759453	171	>40h
s953	16	23	29	347	173009	442145	318026413	78	>50h
s1196	14	14	18	477	240715	625332	298200399	82	>50h
s1238	14	14	18	532	244738	635761	348359262	75	>50h
s1423	17	5	74	462	487725	1202990	213210750	174	118467.99
s1488	8	19	6	663	312822	807264	281019998	80	>50h
s1494	8	19	6	673	376028	971065	200903671	95	>25h
s5378	35	49	179	1389	765927	1930193	400905472	88	>100h
s9234	19	22	228	1958	684603	1751646	682645463	55	>100h
s13207	31	121	669	2719	1819462	4396108	398982797	105	>100h
s15850	14	87	597	3560	899443	2248748	601807846	39	>100h
s35932	35	320	1728	11948	2879804	6700741	26632899	40	>100h
s38417	28	106	1636	9219	1651811	4155933	752396702	27	>100h
s38584	12	278	1452	12400	3331928	8303062	566087105	44	>100h

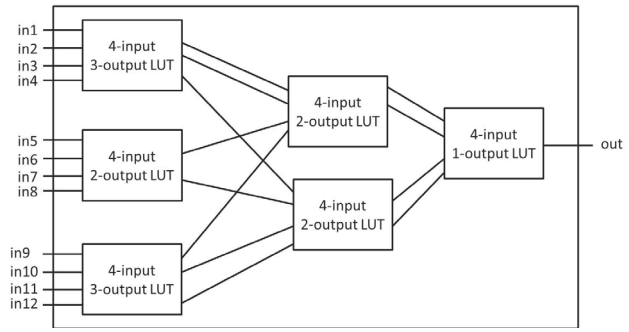
TABLE I
FIXED TOPOLOGY MAPPING WITH LUT

Name	Vars	Clauses	Conflicts	Tests	Time (s)
s27	187	309	24	6	0.01
s298	7271	16662	827	35	0.04
s344	7226	17594	2354	30	0.2
s349	8234	18308	730	35	0.13
s382	12134	26631	918	43	0.17
s386	14208	30735	2772	54	0.21
s400	12418	28691	1517	41	0.13
s444	11554	28741	2216	36	0.1
s510	25663	68806	11058	66	0.52
s526	29752	76137	4523	77	0.32
s641	27264	55077	1897	80	0.15
s713	21224	46485	1639	56	0.13
s820	69613	168271	12831	127	1.15
s832	89826	217392	16966	158	1.84
s953	68247	172198	19507	94	1.74
s1196	127849	312047	16891	155	2.79
s1238	173882	419099	18423	194	2.95
s1423	88218	193894	9134	90	1.14
s1488	122603	274328	12370	131	1.98
s1494	138915	316904	18147	148	3.13
s5378	732113	1656343	2101830	263	547.32
s9234	1585886	3570590	5564626	423	2100.66
s13207	2723282	4987122	711130	474	557.43
s15850	3203371	6933658	18862813	440	3625.81
s35932	3447907	6548092	1070785	173	7709.4
s38417	17657911	35695772	140866184	901	70512.05
s38584	14291557	28562060	1298930	609	6095.87

TABLE II
FIXED TOPOLOGY MAPPING WITH INPUT POLARITY CHANGES

	Only LUT		Polarity then LUT		Ratio (Only LUT=1)	
Name	Tests	Time (s)	Tests	Time (s)	Tests	Time
s27	16	0.05	15	0.03	0.94	0.60
s208.1	147	11497.52	150	2251.33	1.02	0.20
s298	101	13.08	81	17.62	0.80	1.35
s344	90	13.76	60	7.65	0.67	0.56
s349	78	11.28	62	7.98	0.79	0.71
s382	96	8.43	97	11.47	1.01	1.36
s386	243	216.57	166	117.29	0.68	0.54
s400	85	8.47	85	9.53	1.00	1.13
s444	145	35.21	84	13.00	0.58	0.37
s526	192	432.84	151	242.84	0.79	0.56
s641	242	11739.97	224	1433.61	0.93	0.12
s713	272	1959.96	178	630.03	0.65	0.32
s1423	174	118467.99	167	41954.84	0.96	0.35

TABLE III
FIRST APPLY POLARITY BASED METHOD FOLLOWED BY LUT BASED METHOD



Number	1	2	3	4	5	6	7	8	9	10
Time (sec)	66	13	95	71	361	45	15	683	120	22
#Iteration	96	35	113	97	226	77	41	309	142	56

Number	11	12	13	14	15	16	17	18	19	20	Ave.	Max	Min
Time (sec)	29	170	588	854	57	98	109	33	169	119	185.85	854	13
#Iteration	59	160	292	335	97	127	133	69	164	133	138.05	335	35

Fig. 11. Synthesis result targeting Circuit2 for 20 most common logic functions with 12 inputs [9]

benchmark circuits [9]. The paper shows the most frequently appearing twenty functions, and appropriate configurations for all of them have been obtained as shown in Figure 11. This demostrates the powerfulness and usefulness of the proposed methods for relatively small circuit synthesis.

V. CONCLUDING REMARKS

We have presented logic synthesis and formal verification method with LUT formulation and its simplified method with only polarity change. Generality of the former has been demonstrated, although its processing requirements are high for large circuits. The latter can deal with only a subset of cases but can work for circuits having ten thousand gates. We have also shown results on their combined use. As complete sets of test vectors can be generated, simulation-based verification can guarantee 100% correctness.

REFERENCES

[1] K.-H. Chang, I. L. Markov, and V. Bertacco: Fixing design errors with counterexamples and resynthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD)*, 27(1), 2008 .

[2] S. Krishnaswamy, H. Ren, N. Modi, R. Puri. DeltaSyn: An efficient logic difference optimizer for ECO synthesis, *ICCAD*, 2009.

[3] B.-H. Wu, C.-J. Yang, C.-Y. Huang, and J.-H. R. Jiang. A robust functional ECO engine by SAT proof minimization and interpolation techniques, *ICCAD*, 2010.

[4] A. R. Bradley: SAT-Based Model Checking without Unrolling, *VMCAI11*, 2011.

[5] N. Een, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability, *FMCAD*, 2011.

[6] A. Ling, P. Singh, and S. D. Brown: FPGA Logic Synthesis Using Quantified Boolean Satisfiability, *SAT*, 2005.

[7] A. S.-Lezama, L. Tancau, R. Bodik, S. A. Seshia, V. A. Saraswat: Combinatorial sketching for finite programs, *ASPLOS*, 2006.

[8] M. Janota, W. Klieber, J. Marques-Silva, E. M. Clarke: Solving QBF with Counterexample Guided Refinement, *SAT*, 2012.

[9] A. Mishchenko: Enumeration of irredundant circuit structures, *IWLS*, June 2014.

[10] M. Fujita, S. Jo, S. Ono, T. Matsumoto: Partial synthesis through sampling with and without specification, *ICCAD*, Nov. 2013.

[11] R. K. Brayton, A. Mishchenko: ABC: An Academic Industrial-Strength Verification Tool, *CAV*, 2010.