Feasibility Study of DSP Block Mapping Algorithms for FPGAs Utilizing SAT-solver and Top-down ZDD Construction

Takuya Serizawa† Koyo Shibata† Takashi Imagawa‡ Hiroyuki Ochi†

†Graduate School of Information Science and Engineering, Ritsumeikan University
1-1-1 Noji-higashi, Kusatsu, Shiga, 525-8577 Japan

‡College of Science and Technology, Meiji University
1-1 Kanda Surugadai, Chiyoda, Tokyo, 101-8301 Japan

is0415hv@ed.ritsumei.ac.jp, is0358ir@ed.ritsumei.ac.jp, imagawa@meiji.ac.jp, ochi@cs.ritsumei.ac.jp

Abstract— This paper proposes two algorithms to find the exact optimal technology mapping solution(s) for DSP blocks of FPGAs, one using SAT solver and the other using a top-down ZDD construction method. The exhaustive depth-first search algorithm for DSP block mapping by Shibata et al. introduced several complicated rules for pruning and graph partitioning for speeding up. In contrast, the proposed ones are relatively simple. The runtime of the SAT-solver-based method is comparable to that of Shibata et al., and the ZDD-based method can enumerate all optimal solutions.

I. Introduction

LUT-based island-style FPGAs, which appeared in the late 1980s, had been widely used in various fields due to their flexibility. However, many logic blocks and their mutual interconnection were required when implementing arithmetic operations, including multiplication, to FPGA of that age. Thus, there was a large gap between LUT-based FPGAs and ASICs in delay, area, and power consumption. To efficiently implement datapaths of digital signal processing where arithmetic operations such as multiply-add operations frequently appear, many FPGAs having DSP blocks consisting of hard macros including multipliers, have come to be seen[1].

To effectively use the DSP blocks, a technology mapping algorithm is required that finds the optimal mapping of the application circuit using the DSP blocks of the target FPGA architecture. Ronak et al. proposed a greedy algorithm for technology mapping for DSP blocks[2]. Although the algorithm runs extremely fast, the optimality of the solution is not guaranteed because the algorithm depends on the random selection order of nodes to be covered. Shibata et al.[3] proposed an algorithm that finds an exact optimum mapping solution in terms of the number of DSP blocks used and the wiring resources used for interconnecting them. The algorithm conducts an exhaustive search based on a depth-first search from a given data flow graph (DFG) that consists of the arithmetic operators of the design. They experimentally demonstrated that the exhaustive search algorithm successfully found an exact optimal solution that the previous greedy algorithm could not find.

This paper proposes two algorithms, one using a SAT solver and the other using a Zero-suppressed Binary Decision Diagram (ZDD), for implementing an efficient exact algorithm. Both SAT solver and ZDD are powerful tools for finding solutions to a problem that needs exhaustive enumeration of search space, such as the knapsack problem. SAT solver is a program that solves the Boolean satisfiability (SAT) problem. The algorithm for solving the SAT problem first appeared around the 1960s. Still, since the 2000s, the algorithm has made great strides, and SAT solvers are attracting attention as a fundamental technology in various application domains. ZDD [4] is an efficient data structure that represents a combination set. When it was first proposed in the 1990s, it suffered from exponential memory usage for ZDDs on the way to constructing the target ZDD. However, with the development of a new method called the top-down ZDD construction method in the 2010s, the application to various combination problems is now expanding [5].

From the DFG and the structural description of the target DSP block, the proposed algorithms first enumerate all subgraphs in the DFG that can be realized with one DSP block and generate bit vectors representing the subgraphs, followed by applying the SAT solver or ZDD. After that, the proposed algorithm using the SAT solver converts the bit vectors to a CNF formula that the mapping solution should satisfy. In addition, by giving a constraint to specify the upper bound of the number of DSP blocks used and reducing this upper bound until the solution cannot be found, the SAT solver finds one of the solutions with the minimum number of DSP blocks used. On the other hand, the proposed algorithm using ZDD constructs a ZDD representing the solution set of all map-

pings from the bit vectors by the top-down ZDD construction method. It finds the minimum number of DSP blocks used by evaluating the constructed ZDD. It also reconstructs the ZDD that represents only the solutions of the minimum number of DSP blocks used and finds the total number of such solutions.

Our experimental evaluation with a randomly-generated 33-nodes DFG shows that the proposed method using a SAT solver finds one solution with the minimum DSP block used, 20, in 0.77 seconds. The proposed method using ZDD constructs a ZDD of 21592 nodes representing more than 5×10^{16} solutions in 353.32 seconds. It also finds in 0.094 seconds that the minimum number of DSP blocks used is 20, and there are 196 such solutions.

The rest of this paper is organized as follows. Section 2 reviews the target DSP block, the conventional mapping methods, SAT solver, and ZDD. Section 3 proposes our mapping methods, and Section 4 presents the experimental results. Finally, Section 5 gives a summary and describes future works.

II. Background

A. DSP block

DSP block is a hardware block in FPGAs that consists mainly of dedicated circuits for arithmetic operations such as multipliers and adders/subtractors, and that has a certain degree of programmability[1]. Typically, it supports multiply-add operations that frequently appear in the digital signal processing domain and realizes desirable arithmetic units in terms of performance and power efficiency. While there are various DSP blocks due to the variety of design concepts of FPGA vendors and the diversity in the target application domain, the explanations and experiments of the search algorithms in this paper target the DSP block DSP48E1 from Xilinx for simplicity[1]. As shown in Fig.1, DSP48E1 is a DSP block equipped with a low-power pre-adder, a 25×18-bit 2's complement multiplier, a 48-bit accumulator, a pattern detector, registers, and so on[6]. This paper uses a simple model that focuses only on the adder/multiplier of DSP48E1; we do not care about the bit width of registers and signal wires.

B. Technology mapping algorithm for DSP blocks

Technology mapping for DSP blocks is to generate an implementation of a given application circuit using the DSP blocks of the target FPGA architecture. Similar to the LUT mapping, finding an optimal mapping in terms of area, power, and delay is required. This paper mainly focuses on minimizing the number of DSP blocks used.

Ronak et al. proposed a greedy algorithm for technology mapping for DSP blocks[2]. This algorithm uses a

[1]However, the proposed algorithm is also applicable to DSP blocks other than DSP48E1.

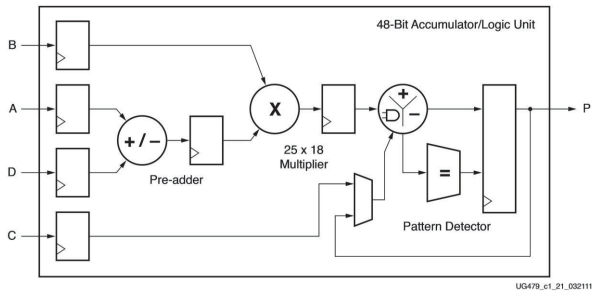


Fig. 1. Xilinx's DSP48E1 architecture[6]

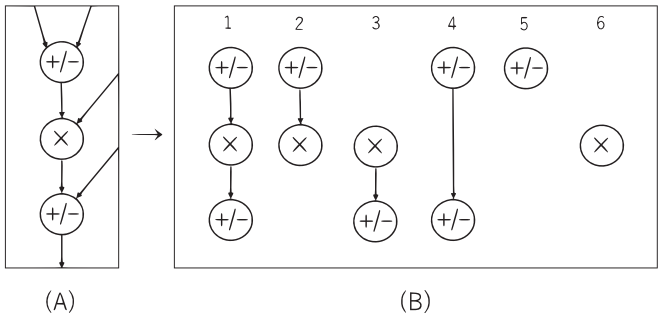


Fig. 2. Generation of template database (TDB)

pre-prepared template database (TDB). TDB is a list of feasible operation patterns by the configuration of the target DSP block. DSP block DSP48E1, the target of [2], is modeled as three arithmetic units connected in series as shown in Fig. 2(A). By choosing enabled arithmetic units from Fig. 2(A), we can find the six arithmetic patterns shown in Fig. 2(B). For the given application circuit, generate a DFG that represents it. After that, repeatedly select a node N randomly from the unmapped nodes in the DFG until all the nodes are covered. At each selection of N, among the largest possible fan-out free subgraphs starting with N, those matching TDB are covered by one DSP block. Ref. [2] also proposed an improved algorithm that preferentially applies the one with a larger number of arithmetic units among the arithmetic patterns in TDB. The problems with their methods include (1) it is specialized for DSP48E1 of Xilinx, (2) it depends on the random selection order of N, and its optimality is not guaranteed, and (3) since mapping by duplicating a DFG node with multiple fanouts is not attempted, it cannot find the optimum solution that [3] can find described later.

Shibata et al. proposed an exhaustive search algorithm that finds an optimal mapping [3]. This algorithm has two notable features as follows.

- It replicates node(s) in a target DFG when it leads to the DSP reduction.

- It extensively applies pruning and DFG partitioning techniques to reduce the runtime while guaranteeing its optimality.

This algorithm also utilizes TDBs but they are automatically generated from a structural description of the target DSP block. The flow of the algorithm is as follows.

- For each node of a target DFG, enumerate subgraphs that terminate at the node.
- Enumerate the subgraph combinations that cover all the nodes in the target DFG with a depth-first search.
- Among the enumerated combinations, find the optimal solution that minimizes the objective function (number of DSPs and their interconnection).

Since the algorithm can only find one mapping even if there are multiple mappings that minimize the objective function, it is difficult to consider other objective functions based on a comparison of multiple optimal solutions. The introduction of new objective functions to the algorithm is not easy due to the complexity of the pruning and partitioning process.

III. Proposed Method

A. Overall Flow

Fig. 3 shows the overall flow common to the two proposed mapping methods. The flow's inputs are the design description of the application circuit to be mapped and the structural description of the target architecture's DSP block. DFG is generated by parsing the former, and TDB similar to Sect. II.B is automatically generated from the latter. Next, the graph substructures (subgraphs) in the DFG that are realizable with one DSP block are extracted by matching TDB to DFG. For each such subgraph g, generate a $3n$-bit vector to represent (1) the node that generates g's output (avail_node), (2) the input source nodes for g that is the intermediate node of the DFG (req_node), and (3) the nodes covered by g (cov_node), where n is the number of nodes in the DFG. For example, the output node of the subgraph g shown by the red ellipse in Fig. 4 is node 4. Thus avail_node of g is expressed by a bit vector 001000 in which the 4th digit is 1. While, g requires input from node 1, and its req_node is expressed as 000001. cov_node of g is expressed as 001010 since it covers nodes 2 and 4. Therefore, the bit vector representation of g is 001000_000001_001010. Mapping by SAT solver or ZDD is performed using the subgraphs expressed by the bit vectors, and the mapping result is output. The solution obtained as a mapping result must satisfy the following two conditions.

1. For each primary output of the DFG, there must be a subgraph that produces it.

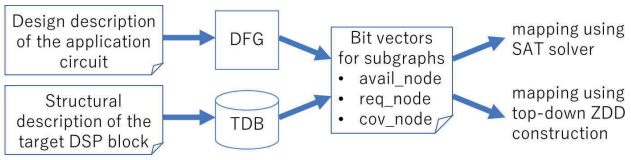


Fig. 3. Overall Flow Common to the two Proposed Mapping methods

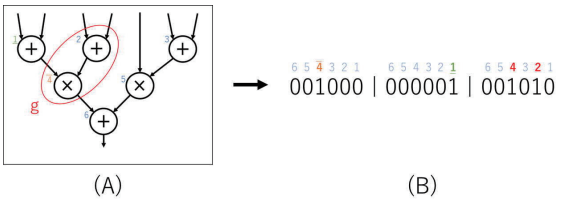


Fig. 4. An Subgraph represented by bit vectors

2. If a source of a subgraph contained in the solution is an intermediate node of the DFG, another subgraph that produces the output of that intermediate node must also be in the solution.

Fig. 5 shows an example of a combination of subgraphs satisfying these two conditions. Node 6 is the primary output node of the DFG. The output of Node 6 is obtained by Subgraph g_1, and the input sources of g_1 are Nodes 2 and 5. The output of Node 2 and that of Node 5 are obtained by Subgraphs g_3 and g_2, respectively.

B. Proposed Mapping Method using SAT Solver

The proposed technology mapping flow using the SAT solver is as follows.

Step 1: From each bit vector expressing a subgraph, generate a CNF formula describing the constraint the mapping solution should satisfy.

Step 2: Give the CNF formula of Step 1 to the SAT solver, and if a solution is found, set a smaller value to the upper limit constraint for the number of DSP blocks used and execute the SAT solver again.

Step 3: Repeat Step 2 until the solution cannot be found (UNSAT).

In Step 1, using the subgraph information (avail_node, req_node, and cov_node) extracted as a bit vector, a CNF formula consisting of the variables corresponding to subgraphs is generated to represent the following constraints.

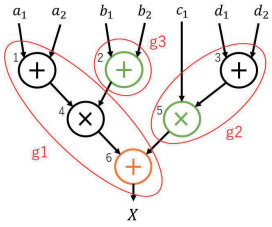


Fig. 5. Subgraph combination conditions

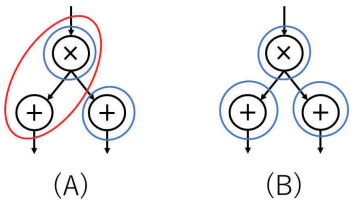


Fig. 6. Redundant subgraph combinations

1. For each primary output node of the DFG, at least one subgraph containing it in avail_node must be adopted.
2. If the subgraph g is adopted, for every "1" bit of g's req_node, it must be included in the avail_node of another adopted subgraph.
3. If the cov_node of the subgraph g_1 is contained in that of g_2, g_1 and g_2 cannot be adopted at the same time.

Constraints 1 and 2 corresponds to the two conditions described in Sect. III.A. While the mapping shown in Fig. 6(A) and in Fig. 6(B) needs the same number of DSP blocks, the latter is better in terms of actually used (i.e., power-consuming) arithmetic components. For this reason, we prune the solution of the former type, as in [3], and introduce Constraint 3 for this purpose.

C. *Proposed Mapping Method using ZDD*

The proposed technology mapping flow using the top-down ZDD construction method is as follows.

Step 1: Considering every edge connecting the subgraphs in the DFG, sort the subgraphs, so that the source subgraph appears earlier than the sink subgraph[2].

Step 2: Construct a ZDD that represents a set of combinations of subgraphs that satisfy the two conditions described in Sect. III.A using the top-down ZDD construction method. Here, ZDD nodes are generated from the root level corresponding to the order of subgraphs determined in Step 1.

Step 3: Extract the solution with the minimum number of subgraphs (= number of DSPs) from the ZDD constructed in Step 2.

In Step 2 above, ZDD is constructed by the top-down ZDD construction method using avail_node and req_node of the subgraphs extracted by the preprocessing described in Sect. III.A. When we use the top-down ZDD construction method, we need to put each ZDD node a variable that represents the search state of that node[5]. In the proposed method, the solution is searched while selecting/not selecting the subgraph according to the order obtained in Step 1. At each ZDD node, the set of DFG nodes whose output was obtained by the subgraphs selected during the ZDD path up to that node is stored in a state variable avail_set. This variable is realized by a bit vector with the same number of digits as the number of DFG nodes, like avail_node and req_node.

Before selecting a new subgraph, check whether all the req_nodes of that subgraph appear in the avail_set obtained so far. (Note that the subgraph selection order is sorted in Step 1, so the selection/non-selection of all subgraphs that can cover the new subgraph's req_node has already been decided.) If all req_nodes of the new subgraph are not included in avail_set, such subgraph is unselectable in this search path. When the subgraph is selectable, introduce a ZDD node that branches with and without selecting the subgraph, and set its 1-branch's avail_set with the union of the subgraph's avail_node and the parent node's avail_set. Fig. 7 shows the pseudo-code of the ZDD node construction function getChild() for implementing the proposed algorithm using the TdZdd package. Manipulation and test of bit vectors are performed by bitwise logic operations. If all primary output of the DFG is covered by avail_set after this operation, the combination of subgraphs selected in this path is a solution.

ZDD constructed by Step 2 represents a set of all solutions, including both optimal and redundant solutions. Therefore, Step 3 reconstructs ZDD by filtering out the solutions whose number of DSP blocks used is non-minimum. Figures 8(a) and (b) show the ZDDs after Step 2 and Step 3, respectively, given DFG6 (to appear in Sect. 4) to the proposed algorithm.

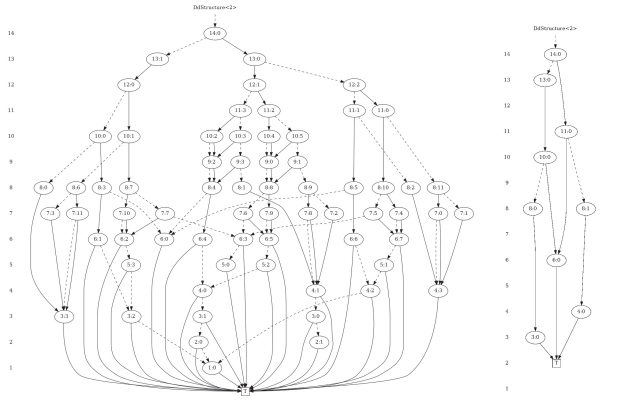
IV. Performance Evaluation

In this section, we compare the mapping runtime and the results by the existing method of Shibata et al.[3] and the two proposed methods described in Section 3. Four DFGs (DFG6, DFG10, DFG20, DFG33) with 6, 10, 20, and 33 arithmetic nodes were used for the evaluation. Each DFG is composed of nodes with addition

[2]We assume that the given DFG is a directed acyclic graph.

```
// n : number of subgraph(s)
// avail_set : set of DFG nodes covered by the output(s) of
//             subgraph(s) selected along the ZDD path so far
// level : level of current ZDD node
//         (numbered in descending order from the root node)
// value : 1 if current subgraph is selected and 0 otherwise
// req_node[n-level] : required node(s) by current subgraph
// avail_node[n-level] : output node(s) of current subgraph
// po : primary output node(s) of the DFG
getChild(avail_set, level, value) {
   if (value == 1) {
      if( (avail_set & req_node[n-level]) != req_node[n-level] ) {
         return 0;
      }
      avail_set |= avail_node[n-level];
   }
   level--;
   if((avail_set & po) == po) return -1;
   return level;
}
```

Fig. 7. getChild() function for the proposed algorithm



(a) after Step 2 (b) after Step 3

Fig. 8. ZDD representing a set of all solutions of DFG6

or multiplication operators, and all nodes have a connection with other nodes or the external inputs/outputs. As the target DSP block, we used the same model as the previous works used (Fig. fig:templatedb(a)), which is a simplified model of Xilinx's DSP48E1. DFG/TDB generation, subgraph extraction, and all processing of existing methods were implemented by Python (ver.3.10.2), syntax analysis was implemented using a Python library PLY (ver.3.11), and graph data manipulation was implemented using Networkx (ver.2.6.3). The SAT-based proposed mapping method was implemented using Python (ver.3.10.2) and Z3-solver (4.8.16.0) Python library. The ZDD-based counterpart was implemented using gcc-9.3.0 (C++ 20) and TdZdd (ver.1.1) C++ library. The time required for each mapping method in this experiment does not include the time for DFG and TDB generation and subgraph extraction; the needed time refers to those after starting the mapping flow using the extracted subgraph until the output of the result.

Tab. I shows the runtime of the program when mapping DFGs by each method, Tab. II shows the mapping result by the SAT-based method, and Tab. III shows the mapping result by the ZDD-based method. The minimum number of DSP blocks used in the optimal solution found by each mapping method was the same. The ZDD size after Step 2 in Tab. III shows the number of nodes of ZDD that represents all solutions, including both the optimal and redundant solutions. The ZDD size after Step 3 in Tab. III shows the number of nodes of ZDD that represents only the optimum solutions in terms of the number of DSP blocks used.

From Tab. I, the ZDD-based mapping method is the fastest for DFG6 and DFG10, but as the number of DFG nodes increases, the runtime increases significantly, and Shibata et al.'s method becomes faster. This is because Shibata et al.'s method took measures against combinational explosions by pruning and graph partitioning, but the proposed method does not at this time. Unlike the ZDD-based mapping method, the runtime increase of the SAT-based counterpart against the DFG size is relatively moderate. It seems comparable to that of Shibata et al.'s method.

From Tab. II, we can observe that the SAT-based mapping method finds a solution speedily if the solution(s) exists. Still, it takes a lot of time to obtain one having the minimum number of DSP blocks by narrowing down the solution using UNSAT judgment. The above results suggest that the SAT solver is excellent for arbitrarily finding one solution but not suitable for finding the best among multiple solutions. At the same time, if the goal of the number of DSP blocks used is given in advance, it can find one solution that meets the criteria faster than the existing method.

From Tab. III, we can see that as the number of DFG nodes increases, the total number of solutions increases explosively, while the total number of optimal solutions is much smaller. This result is an exciting finding suggesting that it is difficult to find and enumerate all optimal solutions in technology mapping using a depth-first search.

Regarding the runtime of the ZDD-based proposed method, (1) the increase in runtime is not so sharp as the increase in the total number of solutions, and (2) the runtime is dominated by the construction of ZDD representing all solutions in Step 2. A solution of more than 5×10^{16} is found from DFG33, and the runtime for mapping solutions increases significantly. However, most of it is the runtime for constructing ZDD for representing all solutions, and the reconstruction of ZDDs representing the optimal solutions only is still very fast. In addition, the number of nodes of the ZDD that represents more than 5×10^{16} solutions is 21592, and the number of nodes of the reconstructed ZDD that represents only 196 optimal solutions is also suppressed to 185. These results are brought by the characteristics of ZDD that enable calculations without solid dependency on the number of solutions. We can confirm that the top-down ZDD con-

TABLE I
Mapping results of proposed and conventional methods

(a) Specifications of DFGs used in the experiment

	DFG6	DFG10	DFG20	DFG33
DFG Nodes	6	10	20	33
Number of Subgraphs	14	20	43	78
Minimum DSP blocks used	3	6	13	20

(b) Runtime for each method[s]

Mapping Method	DFG6	DFG10	DFG20	DFG33
Conventional	0.0046	0.0076	0.014	0.58
SATsolver	0.035	0.040	0.14	0.77
ZDD	0.001	0.005	0.094	354.44

TABLE II
Mapping results of using SATsolver

DFG	Limit constraint for the number of DSP blocks used	Runtime of the SAT decision[s]
DFG6	5	0.0076
	3	0.00091
	UNSAT	0.00030
DFG10	8	0.0077
	6	0.0011
	UNSAT	0.00064
DFG20	13	0.0086
	UNSAT	0.079
DFG33	22	0.010
	21	0.0041
	20	0.11
	UNSAT	0.54

struction method can significantly contribute to speeding up the exhaustive enumeration of solutions.

V. Conclusion

This paper have proposed two exact algorithms to find optimal DSP mappings for FPGAs. The proposed SAT-based method, which is simpler to implement than the previous method, finds the optimal solution in a comparable run-time. The other proposed method utilizing the top-down ZDD construction enumerates all the optimal mappings that minimize the number of DSP blocks, in exchange for an acceptable increase of run-time. In future works, we will introduce (1) the pruning and DFG partitioning techniques employed in the previous method into the proposed method to reduce the run-time, and (2) other metrics in the optimization objective, such as power consumption and signal propagation delay. The proposed algorithms and their improved versions are expected to contribute to the design space exploration of FPGAs with multile types of DSP blocks.

TABLE III
Mapping results of using ZDD

(a) Number of all solutions and optimal solutions

DFG	Number of all solutions	Number of optimal solutions
DFG6	238	4
DFG10	2334	8
DFG20	3976018364	264
DFG33	5236607862923102208	196

(b) Runtime breakdown and ZDD size

DFG	Runtime until Step 2 [s]	ZDD size after Step 2	Runtime of Step 3 [s]	ZDD size after Step 3
DFG6	0.001	74	0.001	9
DFG10	0.004	271	0.001	31
DFG20	0.084	1236	0.005	135
DFG33	353.63	21592	0.094	185

References

[1] Amano, H.: *Principles and Structures of FPGAs*, Springer (2018).

[2] Ronak, B. and Fahmy, S. A.: Mapping for Maximum Performance on FPGA DSP Blocks, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Vol. 35, No. 4, pp. 573–585 (2016).

[3] Shibata, K., Imagawa, T. and Ochi, H.: A Feasibility Study on Realizing General-purpose Technology Mapper for DSPs of FPGAs Using Exhaustive Search, *SASIMI 2021 Proceedings*, pp. 61–66 (2021).

[4] Shin-ichi Minato: Zero-suppressed BDDs for set manipulation in combinatorial problems, *30th International Design Automation Conference (DAC'93)*, pp. 272–277 (1993).

[5] Toda, T., Saito, T., Iwashita, H., Kawahara, J. and Minato, S.: ZDDs and Enumeration Problems: State-of-The-Art Techniques and Programming Tool, *Computer Software*, Vol. 34, No. 3, pp. 3_97–3_120 (2017).

[6] Xilinx Corporation: *7 Series DSP48E1 Slice User Guide*, `https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf` (2011).