Realization of Incompletely Specified Functions in Minimized Reversible Cascades

Manjith Kumar* - kmanjith@yahoo.com, Bala Iyer* - bala.iyer@intel.com, Natalie Metzger* - nmetzger@portgnome.com, Ying Wang* - Ying.Wang@synopsys.com, Marek Perkowski* - mperkows@ee.pdx.edu

**Department of Electrical and Computer Engineering, Portland State University, 1900 SW 4th Avenue, Portland, OR 97201, USA.*

Abstract

There is a need to convert non-reversible functions to their corresponding reversible functions to be realized as reversible cascades. The original MMD (D.M.Miller, D. Maslov, and G.W.Dueck) algorithm for synthesis of reversible functions using cascades of reversible gates [1] can be modified to allow for the inclusion of "don't cares" within the given function's truth table (reversible or irreversible). This was achieved in the approach presented, by first initializing the "don't cares" to binary values, synthesizing the network using the base MMD algorithm, comparing the cost, and iterating to find an implementation with the smallest possible cost. The "don't care" assignment leading to the circuit with the minimal cost can be determined. This paper discusses this algorithm that is an additional module to the MMD algorithm, the results, the pros and cons of using such an algorithm, and future work to improve the proposed algorithm. A heuristic is also covered, that is superior to the traditional backtracking algorithm and can quickly find an MMD solution with minimum cost for circuits that don't have large number of "don't cares".

1. Introduction

Many research groups have discussed the synthesis of reversible networks [1,2,3,4]. The primary applications of reversible networks are low power circuit design, quantum computing [5], and nanotechnology [6].

Few synthesis methods have been proposed for reversible logic. Some of the reported methods are: transformation-based synthesis [1,7,8]; using Toffoli and Maitra-like gates to implement an EXOR sum of products (ESOP) and wave cascade [9]; exhaustive enumeration [10]; search using Reed-Muller representation [4]; heuristic methods that iteratively make the function simpler (simplicity is measured by the Hamming distance [11] or by spectral means [12]).

Most of the reported synthesis methods work only on completely specified reversible logic functions. As of yet, "don't cares" cannot be handled efficiently by these methods. The problem of synthesizing incompletely specified reversible functions is of importance in many areas like 1) the design of incomplete oracles for machine learning applications 2) design of blocks included in larger oracles or spectral transforms where "don't cares" occur similar to classical logic network design and 3) reversible state machine design and quantum automata design. For instance "don't cares" are introduced when realizing the excitation functions of flip-flops such as JK, SR, jk or sr. Another cause of occurrence of "don't cares" in state machines is encoding of states. For example, five states encoded in three input signals create three columns of "don't cares" in the transition table. Five internal states encoded in three memory elements create three rows of "don't cares". Similarly "don't cares" are common in both synchronous-like and asynchronous-like realizations of reversible and quantum automata.

More "don't cares" can occur in combinational logic when the number of output lines is smaller than the number of input lines. In such a situation, ancilla bits are added on the output side to maintain the reversible nature. This ancilla bit addition will manifest itself as a column of "don't cares" added to the output part of the truth table.

In this paper, techniques will be presented to synthesize reversible networks with these "don't cares" output values. The presented algorithm is incorporated to the "transformation-based" reversible circuit synthesis software developed by D.M.Miller, D. Maslov, and G.W.Dueck (MMD) [1].

The proposed algorithm that will be referred to henceforth as "Don't Care Algorithm for Reversible Logic" (DCARL), can be simplified to three steps:

Step 1: Assign values to the "don't cares" outputs, and map the outputs according to the assigned input value.

Step 2: Use the MMD algorithm to synthesize the network.

Step 3: Compare the cost in terms of the number of Toffoli gates and keep track of the "don't cares" values with the minimal cost. Backtrack to find *K* solutions or until no more backtracking is possible.

2. The Don't Care Algorithm for Reversible Logic (DCARL)

The function in MMD is represented as a binary truth table, however, in DCARL, there is an additional representation for "don't cares". The "don't cares" are represented by the character (-) or (x) as a placeholder for future "don't cares" assignments that take place in DCARL process. This "don't cares" symbol does not initially have any binary value assigned to it. A branching/ backtracking approach is used to assign a binary value to the "don't cares" in DCARL. This approach results in the "don't cares" assignment evaluation continuing to be a two-valued evaluation. The inputs continue to be on the left hand side and the outputs remain on the right hand side of the truth table and the cascade under implementation. The function is evaluated from the output to the input since the reversible circuit can be constructed from either end [1]. Similar to the classical MMD algorithm, for every assignment of a "don't cares", reversible gates (Fredkin, Toffoli, etc.) are applied at the end of the circuit. It is assumed that all gates used (NOT, CNOT, Toffoli, Fredkin) are self inverses.

The DCARL is designed as a "module" that is completely separate from the MMD code. It uses the MMD code with the standard binary MMD format created by DCARL (see Figure 2-1).

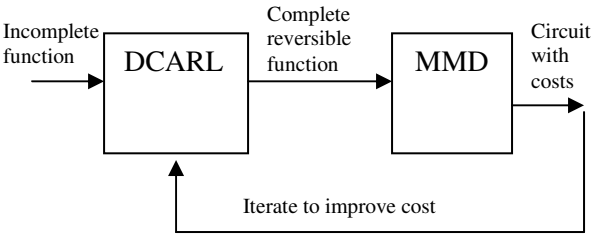

Figure 2-1 – Black box model of the system. Input and output of DCARL in the form of truth tables, incomplete and complete, respectively.

The original MMD algorithm works by taking a given binary truth table and for each truth table row, adding Toffoli and Fredkin gates one at a time such that the output equals the input. Then for each truth table row, MMD checks to make sure the added gates do not affect the previous outputs. This cycle continues until it has iterated through all values in the truth table.

There are currently two versions of the DCARL that will be discussed in this paper. The conventional DCARL which is the algorithm referred to by "DCARL" and a Tcl (Tool Command language) based heuristic version which is referred to by "HDCARL."

2.1 Complexity of DCARL

For the worst possible circumstances, the complexity for the DCARL can run into $O(2^M)$, where M is the total number of "don't cares" in the truth table of the function (Its assumed that M >> n) . However the DCARL will run much faster as the number of unassigned "don't cares" decreases over each iteration.

2.2 DCARL method for constructing a reversible function

Step 1. For a function with n input variables, write all the outputs as a binary truth table in the natural order of the 2^n input minterms.

Step 2. If the function has the same completely specified output for more than one minterm (eg: if F (x'yz) =F (x'y'z) =001), then add a column of "don't cares" to the most significant position of the outputs.

Step 3. Going from top to bottom in the truth table, assign values to the "don't cares" bits in the first incompletely specified minterm in the truth table. The number of possible values for 'm' "don't cares" bits in an 'n' bit minterm is 2^m, ranging from all zeros to all ones. Initially, all zeros are assigned to the "don't cares" bits. When needed, increment the "don't cares" bits in steps of one.

Step 4. Once a value is assigned to all the "don't care" bits of a term, go back and check all previous terms to ensure that the assigned "don't care" combination is unique (see definition of unique assignment below).

Step 5. If unique, go to the next minterm and check for uniqueness. Wherever "don't care" bits are encountered, they are first assigned certain values before the uniqueness is checked.

Step 6. If not unique, increment the value of the "don't cares" bits by 1 (i.e. the least significant "don't care" bit is set to 1 and the others are set to 0). If it still does not give a unique value, then increment it again. Keep incrementing until a unique value is found.

Step 7. If none of the 2^m bit combinations in a 'n' bit value with 'm' number of "don't cares" gives a unique value, then add a column of "don't cares" to the most significant position of the outputs and try again from S3.

Step 8. If at any point, it was found that a previously assigned "don't care" combination conflicts with a "no-other-choice" combination (see definition of "no-other-choice" assignment below) that comes down the truth table at a later instance, backtrack to the conflicting assignment and re-assign the "don't cares" with another combination. Start the process again from Step 5.

Step 9. When all the minterms have been assigned values such that each value is unique, and in adherence with the original incompletely specified function, stop the process.

Definition 1: *"No-other-choice" assignment*: If the 'm' "don't cares" in an 'n' bit value are such that only one of the 2^m bit combinations will give a unique, non-repeated value to the n bit combination (i.e. all other 2^{m-1} combinations give an existing completely specified output) it is considered a "no-other-choice" assignment.
E.g.: if F (xyz') =001 and F (xy'z') =00X, the only possible value for the "don't care" X is 0. Note that if F(xyz')=00X and F(xy'z')=00X, then the assignment is not "no-other-choice" because F(xy'z') could also take the value 000 or 001 depending on whether F(xyz') is 000 or 001

Definition 2: *Uniqueness of "don't care" assignment:* For all outputs, no two outputs are the same for a completely specified reversible function. When assigning binary values to a given "don't care" symbol, the resulting output should be different from all previously assigned and all completely specified output values.

3. Formalisms and examples for DCARL

Let F(x, y, z) be defined by Table 3-1.

Table 3-1 – Truth Table of the initial specification of the incompletely specified 3*3 function

P, Q, R = F(x, y, z).

Inputs (xyz)	Outputs (PQR)
000	0XX
001	010
010	00X
011	1X1
100	1X1
101	1XX
110	1XX
111	00X

The task of the DCARL is to convert outputs PQR to completely specified binary values so that F becomes reversible, i.e. one-to-one mapping.

In the Table 3-2, black color indicates a newly assigned bit combination that is repeated i.e. it appears previously in the truth table. Gray represents an assignment that is temporarily valid (i.e. it leads to a value that hasn't been used till now, but could turn out later to be invalid as the DCARL moves further down the table). Observe that similar to the MMD algorithm, the output vectors in Table 3-2 become fixed from top to bottom and a once fixed output value is never modified. This is seen as part of each S_i column above the grey level row.

Table 3-2 – DCARL outputs, steps S1-S5.

S1	S2	S3	S4	S5
0XX	000	000	000	000
010	010	010	010	010
00X	00X	000	001	001
1X1	1X1	1X1	1X1	101
1X1	1X1	1X1	1X1	1X1
1XX	1XX	1XX	1XX	1XX
1XX	1XX	1XX	1XX	1XX
00X	00X	00X	00X	00X

Column S1 is just the outputs (PQR) column from table 3-1. For the first pass of column S1, DCARL finds the first output that contains "don't cares". Then DCARL assigns binary values to the two "don't care" symbols, as detailed in section 2.2. It checks this new assignment with the expected assignment and finds that it is valid, as illustrated in column S2 of Table 3-2. DCARL then continues down the truth table outputs until another assignment can be made. When it finds the next output that has "don't cares" it applies the same steps as in S2. Column S3 indicates in Table 3-2 that the bit combination 000 has already been used.

Table 3-3 – DCARL outputs, steps S6-S10.

S6	S7	S8	S9	S10
000	000	000	000	000
010	010	010	010	010
001	001	001	001	001
101	101	101	101	101
101	111	111	111	111

1XX	1XX	100	100	100
1XX	1XX	1XX	100	101
00X	00X	00X	00X	00X

Table 3-4 – DCARL outputs, steps S11-S15.

S11	S12	S13	S14	S15
000	000	000	000	001
010	010	010	010	010
001	001	001	000	00X
101	101	101	1X1	1X1
111	111	111	1X1	1X1
100	100	100	1XX	1XX
110	110	110	1XX	1XX
00X	000	001	00X	00X

Partial assignment S13 proves that one of earlier assignments was invalid, because there are no possible assignments to 00X that could lead to unique Input/Output mapping. Thus, the first instance of backtracking occurs at S14. The backtracking to the possible invalid assignment occurs and the re-assignment is performed as shown in S14 (Table 3-4).

When DCARL backtracks, it references the data structure for the original output values before they had been assigned binary values. It then restores the original output values to the last output that had "don't cares". DCARL then tries a new combination. If that new combination also leads to an invalid assignment, DCARL backtracks to an earlier point in process. If that does not work, DCARL backtracks to another possible assignment as shown in S15 (Table 3-4).

Table 3-5 – DCARL outputs, steps S16-S20 with backtracking.

S16	S17	S18	S19	S20
001	001	001	001	001
010	010	010	010	010
000	000	000	000	000
1X1	101	101	101	101
1X1	1X1	101	111	111
1XX	1XX	1XX	1XX	100
1XX	1XX	1XX	1XX	1XX
00X	00X	00X	00X	00X

Table 3-6 – DCARL outputs, steps S21-S24.

S21	S22	S23	S24
001	001	001	001
010	010	010	010
000	000	000	000
101	101	101	101
111	111	111	111
100	100	100	100
100	101	110	110
00X	00X	00X	000

Table 3-7 – DCARL outputs, steps S25-S29 with backtracking.

S25	S26	S27	S28	S29
001	010	010	011	011
010	010	010	010	010
000	00X	00X	00X	000
101	1X1	1X1	1X1	1X1
111	1X1	1X1	1X1	1X1
100	1XX	1XX	1XX	1XX
110	1XX	1XX	1XX	1XX
001	00X	00X	00X	00X

In S26, DCARL backtracks again and re-assigns a value to the first minterm (Table 3-7).

Table 3-8 – DCARL outputs, steps S30-S34.

S30	S31	S32	S33	S34
011	011	011	011	011
010	010	010	010	010
000	000	000	000	000
101	101	101	101	101
1X1	101	111	111	111
1XX	1XX	1XX	100	100
1XX	1XX	1XX	1XX	100
00X	00X	00X	00X	00X

Table 3-9 – DCARL outputs, steps S35-S38.

S35	S36	S37	S38
011	011	011	011
010	010	010	010
000	000	000	000
101	101	101	101
111	111	111	111
100	100	100	100
101	110	110	110
00X	00X	000	001

At the end of stage 38 (Table 3-9), a completely specified reversible function F(x, y, z) is found as shown in Table 3-10.

Table 3-10 – Original function versus final function F(x, y, z) after DCARL application.

Inputs (xyz)	Original Outputs	Final

	(PQR)	Outputs (PQR)
000	0XX	011
001	010	010
010	00X	000
011	1X1	101
100	1X1	111
101	1XX	100
110	1XX	110
111	00X	001

After obtaining the final truth table (see Table 3-10), this truth table is then provided as the input data to the MMD algorithm to synthesize the network.

4. Experimental results

For testing, the DCARL was applied against six-bit and nine-bit incompletely specified reversible functions. Some of these benchmarks for testing were obtained from Dmitri Maslov's Reversible Logic Synthesis Benchmarks Webpage [13]. The other incompletely specified functions were generated using an "Incompletely Specified Function Generator" program developed by our group and posted on the webpage [14]. This program creates incompletely specified functions of any number of variables using random number generators. Multiple versions of these functions were tested. The cases included 20%, 40%, 60% and 80% "don't cares" in the function outputs. When generating a function with higher percentage of "don't cares", the already existing "don't cares" were preserved. The successive increase in the number of "don't cares" in the given function was expected to allow the DCARL to find solutions with smaller costs. This expectation was tested by analyzing the MMD cost that was produced from each of the runs.

For each benchmark function and each percentage of "don't cares," DCARL produced up to a maximum of 25 completely specified functions. Out of those 25 completely specified functions, the completely specified function with the lowest MMD cost was selected. Figures 4-1 and 4-2 show the trends found in this testing.

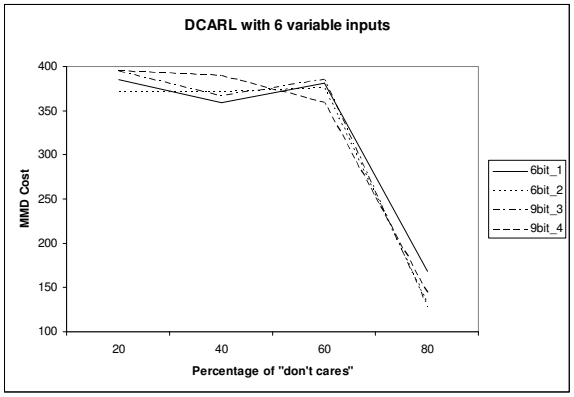

Figure 4-1 – DCARL trend of MMD cost versus percentage of "don't cares" for four 6-bit benchmark functions.

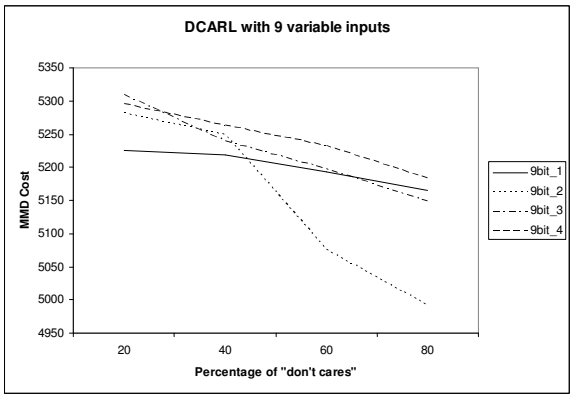

Figure 4-2 – DCARL trend of MMD cost versus percentage of "don't cares" for four 9-bit benchmark functions.

The goal was to see if DCARL could allow MMD to handle incompletely specified functions. The current implementation of DCARL doesn't guarantee an optimal solution with the absolute minimum cost. Also, these tests were an exploration into whether the MMD cost of the incompletely specified function could be improved upon through multiple iterations of DCARL. In the worst case, finding even a non-optimal solution may take a huge amount of iterations, because of the possibility of large number of collisions and a proportionate increase in backtracking. DCARL opts for a tradeoff by adding a column of "don't cares" if the number of iterations exceeds a maximum limit. This addition of an ancilla bit guarantees a quick solution.

The cost of a solution generated by DCARL was expected to decrease with an increase in the percentage of "don't cares" in the input. This expectation was based on the greater flexibility in bit assignments associated with more "don't cares". But

the graphs do not always show a monotonically decreasing trend because in many cases, the solution with lower cost was beyond reach since we have set limits to the number of iterations.

5. An alternate heuristic for finding optimal MMD assignment (HDCARL)

The problem with the conventional DCARL is that it relies on an initial seed. If the seed chosen (first assignment) is not optimal then the conventional DCARL does not produce the circuit with minimum cost. Iterating through all possible combinations in the "don't care" space is very expensive and computationally intensive. Therefore, the development of a heuristic that can produce a large number of assignments for a given "don't care" set was undertaken. This algorithm is called HDCARL or heuristic "don't cares" algorithm for reversible logic. It can quickly provide cost analysis for each one of those assignments. The assignments that are based on the HDCARL are non-greedy by nature so it has the capability to produce optimal results.

However the HDCARL has a complexity of $O(2^M)$ where M is the total number of "don't cares." Also, the HDCARL does not work for non-reversible functions at this point in time. For non-reversible functions, a column of "don't cares" needs to be added. However, the current HDCARL, because of its $O(2^M)$ complexity, will not assign the "don't care" column until it completes all iterations. A quick enhancement could be to loop to a finite value and, in the absence of a solution, append a column of "don't cares" (assuming that the function is not reversible). Another option could be to have a user input that provides the number of "don't care" columns that are needed for the circuit.

The HDCARL includes the following steps:

1) Concatenate all the output patterns (000, 001, 010, 01x, 11x for example) into one string.
2) Find out all patterns that have no "don't cares" in them (000,001,010) and mark them in a hash table as a "required pattern".
3) Count all the "don't cares" in the circuit.
4) For each of the "don't cares" in the circuit iterate from 0 to M where M is the total number of "don't cares" and assign them in natural order. i.e. from 0 to 2^M.
5) Split the string into the output patterns of original width and find out if there are any collisions between the output patterns. A collision is defined as a pattern repeated more than once.
6) If no collisions are found, run MMD on the pattern, calculate cost and store this cost in the hash table.
7) Output the pattern that has the least cost.

However, the pattern has 2^M number of "don't cares". For 36 "don't cares", the HDCARL executes 2^{36} number of iterations. This causes an exponential increase in the runtime of the HDCARL program. However, the program does provide the last minimum cost that it detects. Another enhancement could be to break the program sequence after a finite number of iterations. For guaranteed optimality all the iterations must be executed. These iterations, however, take very little time to run, and a circuit with 2000 iterations (11 "don't cares") runs in 10 seconds.

Table 5-1 – Number of output patterns as a function of percentage of "don't cares" for the 8-bit Gray code benchmark from [13].

Percentage of "don't cares" input to HDCARL	Number of "don't cares" input to HDCARL	Number of output patterns generated
5	2	4
10	4	16
15	7	96
20	9	384
25	11	1536

Table 5-1 was based upon 8-bit gray code Maslov benchmark [13] where the "don't cares" were introduced to the input of HDCARL. It illustrates that the number of outputs produced by HDCARL increases dramatically as the percentage of "don't cares" in the applied function to HDCARL increases.

6. Conclusions and future research

Tests have confirmed the expectation of improved MMD costs when the percentage of "don't cares" included in a given function increases for the 9-bit and 6-bit functions. This was presented here. The functions with fewer variables did not provide conclusive evidence of this property. This is because of the non-optimal strategy of assigning values to "don't cares" or is a result of the non-optimal backtracking mechanism. The algorithm tried to assign output bit values that were very close to their corresponding input bit values, in accordance with the MMD method. But this may not always yield circuits with minimum cost. More testing should be done to confirm or deny this finding in a more conclusive manner.

Our research was successful in proving that it is possible to add a preprocessing module to MMD so that incompletely specified functions can be realized in reversible cascades.

DCARL program, iterating through the "don't care" cases, is by no means optimal. Also, since the MMD uses truth tables as the form in which to represent functions, the ability of the software is slowed by the size of data that this search method encounters. Maslov, Miller and Dueck have pointed out that their code is in need of some optimization [1]. The scope of this project did not address this optimization problem, it only added to the functionality of the MMD. Therefore, further work needs to be done to speed up the code execution by better representation of data.

As of writing this paper, DCARL had not been incorporated into the MMD source code. It worked as a separate module that could be run integrated with the MMD program (see Figure 2-1). Additional work could also be done by applying some sort of intelligent cost analysis within DCARL. This method may then be able to generate circuits with lower MMD cost.

Acknowledgements:

The authors would like to express their sincere thanks to Dr. Michael Miller and Dr. Dmitry Maslov for providing their MMD application for our work. We also appreciate the tremendous effort put in by Mr. George Opsahl, formerly of Mentor Graphics, in evaluating the software.

References

[1] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in Proc. *DAC*, Anaheim, CA, p. 318, June 2-6, 2003.

[2] P. Kerntopf. "A new heuristic algorithm for reversible logic synthesis," in Proc. *DAC*, pp. 834-837, June 2004.

[3] D. Maslov and G. W. Dueck. "Reversible cascades with minimal garbage," IEEE Transactions on CAD, 23(11): pp.497-1509, November 2004.

[4] A. Agrawal and N. K. Jha. "Synthesis of reversible logic," in Proc. *DATE*, Paris, France, pp. 710- 722, February 2004.

[5] M. Nielsen and I. Chuang. "Quantum Computation and Quantum Information," Cambridge University Press, 2000.

[6] R. C. Merkle. "Reversible electronic logic using switches," Nanotechnology, 4:21-40, 1993.

[7] K. Iwama, Y. Kambayashi, and S. Yamashita, "Transformation rules for designing CNOT-based quantum circuits," in Proc. *DAC*, New Orleans, LA, pp. 419-424, Junne 10-14, 2002.

[8] D. Maslov, C. Young, D. M. Miller, and G. W. Dueck. "Quantum circuit simplification using templates," in Proc. *DATE*, Munich, Germany, pp. 1208-1213, March 2005

[9] A. Mishchenko and M. Perkowski, "Logic synthesis of reversible wave cascades," in Proc. Int. Workshop Logic Synthesis, pp. 197-202, June 2002.

[10] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Reversible logic circuit synthesis", in Proc. Int. Conf. Computer-Aided Design, San Jose, CA, pp. 125-132, November 10-14, 2002.

[11] G. W. Dueck and D. Maslov, "Reversible function synthesis with minimum garbage outputs," in Proc. 6th International Symposium on Representations and Methodology of Future Computing Technologies, Trier, Germany, pp. 154-161, March 2003.

[12] D. M. Miller and G. W. Dueck, "Spectral techniques for reversible logic Synthesis," in Proc. 6th International Symposium on Representations and Methodology of Future Computing Technologies, Trier, Germany, pp. 56-62, March 2003.

[13] D. Maslov, G. W. Dueck, and N. Scott, "Reversible Logic Synthesis Benchmarks," [Online document], Available HTTP: http://www.cs.uvic.ca/~dmaslov/, November 15, 2005 [cited December 5, 2006].

[14] Manjith Kumar's webpage, Available HTTP: http://web.cecs.pdx.edu/~kmanjith/, January 22, 2007 [cited January 22, 2007].