Design and Design Automation of Rectification Logic for Engineering Change

Cheng-Hung Lin*, Yung-Chang Huang*, Shih-Chieh Chang* and Wen-Ben Jone**
*CS Department, National Tsing Hua University, Hsinchu, Taiwan
**ECECS Department, University of Cincinnati, Cincinnati, Ohio, USA

Abstract

In a later stage of a VLSI design, it is quite often to modify a design implementation to accommodate the new specification, design errors, or to meet design constraints. In addition to meet the design schedule for the new implementation, the reduction of the mask set have become very critical. In this paper, we propose a new method to add a programmable rectification module to reduce the mask cost and to improve the turn around time. When a modification is needed, one can program the rectification module to achieve the new implementation. The rectification module can be designed by one mask programmable gate array, or an embedded FPGA. To reduce the size needed for the rectification module, we also propose algorithms, which can intelligently select some internal signals of the old implementation to become pseudo primary inputs and primary outputs. Our experimental results are very encouraging.

1. Introduction

In a later stage of a VLSI design, frequently a designer may need to modify a design implementation to accommodate new specification, design errors, or to meet design constraints such as timing, and power consumption constrain. To meet the design schedule, it is desirable to have small modification of the original design. The problem of obtaining small modification for a new implementation is referred to as the engineering change problem. Recently, in addition to engineering effort, the reduction of mask sets has become critical issue for the new engineering change problem due to the dramatic price increase of a mask set. Traditionally, to reduce the mask set cost, a designer may randomly insert spare gates scattering inside a design so that when a re-implementation is required, the new implementation may be achieved by using spare gates. However, most of time, three masks or more masks are still needed for a change.

Unlike the ad-hoc approach of adding spare gates for engineering change, we propose an organized way of inserting a (programmable) rectification module into a circuit for reducing the mask cost in this paper. The rectification module can be programmed to accomplish the new implementation. Figure 1 shows an example adding a programmable rectification module together with the original implementation. The inputs of the rectification module and the original implementation are connected together and the outputs are connected to inputs of XOR gates. Initially, Boolean functions of the rectification module are programmed to "0" so outputs of the whole design (with the rectification module) are the same as the original implementation. The programmable rectification module can be an FPGA, or one mask programmable PLA/ROM while the original implementation can be any design style such as the standard cell design or full custom design. When re-implementation is required, designers can use one or no mask to modify the rectification module to accomplish the new implementation.

The insertion of rectification module can increase the circuit area. Therefore, it is important to have a small size of the rectification module to reduce the overhead. However, depending on the engineering change required for the new implementation, one may need large or small rectification logic. The reason for potential large rectification logic is that the rectification module is "correlated" only through the primary inputs and outputs. As a result, an "error" in an internal node of the original implementation may sometimes need large logic to rectify due to the need for controllability and observability of the error. Since the rectification

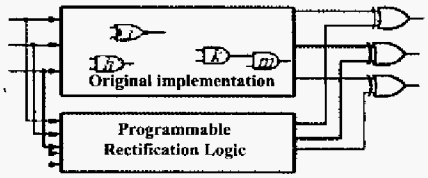

Figure 1 Programmable rectification logic

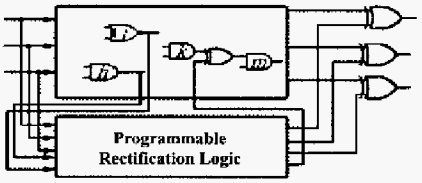

Figure 2 Rectification logic with inputs and outputs from internal nodes

module has to be inserted before knowing what will be changed, if the size is too small, it may not achieve the re-implementation, or if the size is too big, it may waste area. In this paper, we have proposed a new architecture, which can intelligently select some internal signals from the original implementation as inputs or outputs of the rectification module so as to improve the controllability and observability of the rectification module in Figure 2. In the figure, the inputs of the rectification module include primary inputs and some internal signals. And the rectification module can modify both the primary outputs and internal signals of the original implementation. To avoid large number of inputs and outputs for the rectification module, we design input-select and/or output-select modules, which can be programmed to select only partial of all the inputs/outputs. The input-select/output-select modules should be designed similarly to the rectification module in the way that they are either one mask programmable logic or embedded FPGA.

In many designs, data path circuits such as multipliers and adders are not likely to be changed in most cases of the engineering change. Many design modifications occur in the control logic. Therefore, with proper design knowledge, the rectification module should be inserted appropriately at the places where design change is anticipated. In comparison to the use of an FPGA for the entire design (not for the engineering change), the FPGA design style can be applied to the cases where large amount of design change is possible but may require significant more area and delay penalty than the use of a rectification module. Note that the delay penalty can be just an XOR gate delay for the rectification method as in Figure 1 shown. The objective of the rectification module targets at the cases where only minor modification of the original implementation is needed and therefore may not be suitable for the cases when significant change is needed.

2. Previous Work on Engineering Change

According to how the rectification logic is added, we can divide previous works into two categories. One allows the rectification logic mixed with the internal logic while the other requires not

changing the old implementation and extra logic is added in the location before or after the old implementation. In [2], the error diagnosis is first done and the rectification is performed by matching a pre-defined error type. A structural approach [3] establishes the correspondence information between the old implementation and the new specification, and then replaces the pieces of logic in the new specification with pieces of logic from the old implementation. A re-synthesis based approach [4] applies a symbolic error-diagnosis technique to find a single-fixable signal. And then the function of the single-fixable signal is replaced by a new function. In their approach, a primary output partition algorithm is employed to find a single-fixable signal. In [5], a hybrid approach attempts to reduce the potential error region by using local-BDD based verification technique. Then, symbolic error-diagnosis techniques are applied to partially and incrementally correct the old implementation. Structural and re-synthesis based approaches are integrated in the approach. Also, in [6], they consider the problem of correcting errors in a macro-based circuit. Both accurate correction procedures and heuristics are given for correcting multiple errors. The above approaches allow the internal logic of the old implementation to be modified. In [5], they attempt to keep the whole old implementation unchanged. The pre-logic or post-logic are inserted for rectification.

3. The Rectification Module

In this section, we discuss the structure of a rectification module. We assume that the error diagnosis has been performed and the correction of errors is represented by the new implementation. For simplicity, let us consider one output at a time. Let a *differentiating vector* for an output, *out* be an input combination which when simulated, will produce different values at output *out* in the old and new implementation. Consider the example in Figure 3 where Figure 3(a) shows the original implementation and Figure 3(b) shows the new specification. In the example, gate i_3 which is an OR gate in the original implementation is corrected to become an AND gate in the new implementation. Vector v_1 = (000111) is a differentiating vector for output g because simulating v_1, the original implementation has the output of g=1 while the new specification has g=0.

Note the outputs of the original implementation and the rectification module are connected to XOR gates. To correct an output, if an input combination is any among those differentiating vectors for the output, the rectification module will need to produce a value 1 to correct the error; otherwise, it outputs a value 0 to preserve the original function. Therefore, the on-set of the rectification function are those differentiating input combinations. For the same example, we can find that input combinations (a, b, c, d, e, f) = {(000111), (010111), (100111)} can distinguish between both circuits' primary output g and are all differentiating vectors for g. Therefore, we can construct the rectification logic by using all g's differentiating vectors as the on-set of the function which is $b'c'def + a'c'def$. We can also find that the rectification logic for primary output h is $b'c'de + a'c'de$. The rectification logic is shown in Figure 3(c).

If the rectification module can use some internal signals as inputs, the rectification function may be greatly simplified. Let us consider the same example in Figure 3. If internal node i_1 can be an input of the rectification module in Figure 3(d), the rectification logic can be reduced to $i_1'def$ for output g, and $i_1'de$ for output h. By using internal signals as inputs, we can improve the controllability of the rectification module. We can further reduce the complexity of the rectification logic by "correcting" an internal signal in Figure 3(e). In the example, an extra XOR gate is added in the fanout of gate i_4. In this way, the rectification logic can be reduced to $i_1'de$.

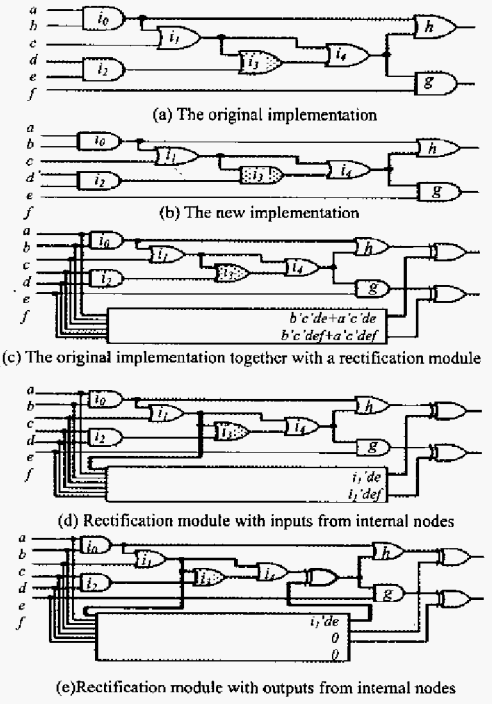

Figure 3 An example of the engineering change.

4. Algorithms for Choosing Proper Internal Nodes as Inputs/Outputs

By choosing internal signals as new inputs or outputs, we can reduce the size of rectification module. However, the rectification module is designed before we perform any engineering change. Since adding internal signals as inputs or outputs can increase the area/delay overhead of the design, it is important to select proper internal signals to achieve high quality. Let those inputs pulled out from internal signals be *pseudo inputs* and those internal signals which input to XOR gates with the rectification logic be *pseudo outputs*. Pseudo outputs are those internal signals which can be corrected by the rectification module. In Figure 3(e), node i_1 is a pseudo input and i_4 is a pseudo output.

Node x is said to *feed* node y (node y is said to be fed by node x) if there exists a directed path from node x to node y. The set of all primary or pseudo primary inputs nodes that feed a node is called its *dependency set*. The cardinality of the set gives the *dependency number* of the node.[7] In Figure 4, the dependency set of node g is $\{a, b, c\}$ and its dependency number is three. If a node has a large dependency number, we may need a large size of rectification logic to implement the node's function. Reducing dependency number of each node under a proper value can directly reduce the size of rectification logic. Therefore, our idea is to choose a set of internal nodes as pseudo inputs/outputs so that the dependency number of each node is less than or equal to k. The value k is set to 15 in our experiments. (We will discuss the effect of k later.) We describe an efficient procedure based on the concept in [7] as follows.

The set of all nodes that have their dependency no greater than k is called the *good set*, otherwise the *bad set*. In Figure 4, the dependency number of nodes f, g, h, i, j, k and l are 2, 3, 2, 5, 5, 5 and 5 respectively. Let k be 3. The good set and bad sets for this circuit are $\{f, g, h\}$ and $\{i, j, k, l\}$, respectively. If a good node is chosen as a pseudo input/output, the dependency number of other

nodes will be changed. Intuitively, we would like to choose a node to be a pseudo input/output so that many bad nodes will be become good nodes. If we choose node g as pseudo input/output node, the dependency number of nodes f, g, h, i, j, k, l become 2, 3, 2, 3, 3, 3 and 3 and all nodes are in good set. In our algorithm, we recursively choose a node from the good set in a way that as many as possible bad nodes will become good nodes till all the nodes become good nodes.

5. Algorithms for Finding the Rectification Logic

In this section, we discuss how to obtain the rectification logic to rectify errors. Note that as mentioned in the Section 2, the on-set of the rectification logic for an output is the set containing all differentiating vectors for the output. Also in our implementation, we allow don't care variables in a differentiating vector. For example, if an input vector (a,b,c) =(0,0,*) is a differentiating vector, both (a, b, c) = {(0,0,0), (0,0,1)} are differentiating vectors.

Our algorithm consists of two steps to find rectification logic. The first step attempts to find "faulty" regions of errors. The faulty regions are those internal circuit elements surrounding the errors and are obtained in the following. We start from an error site and traverse its transitive fanouts till reach pseudo/primary outputs. Then from those pseudo/primary outputs, we traverse their transitive fanins till reach pseudo/primary inputs. The regions surrounded by the reached pseudo/primary inputs and pseudo/primary outputs, are faulty regions. Consider the same example in Figure 3 where the error occurs in node i_3. We traverse the fanouts of i_3 and reach a pseudo output node i_4. Then from node i_4, we traverse toward the primary inputs and reach pseudo/primary inputs $\{i_1, d, e\}$. Therefore the faulty region contains all the circuit elements which are surrounded by nodes $\{i_4, i_1, d, e\}$. The faulty region is highlighted by the dotted circle in Figure 6(a). If there are multiple errors, we may have several faulty regions. The reason of obtaining the faulty regions is that the rectification logic only needs information in the faulty regions to correct errors. It is because that those pseudo /primary inputs and pseudo/primary outputs surrounding the faulty regions are enough to have the controllability and observability for correcting errors. Therefore, to obtain all differentiating vectors for all outputs, we can conceptually remove all other portions of the circuit and keep only the faulty regions to reduce the computational complexity.

After obtaining the faulty regions, in the second step, we iteratively apply an ATPG algorithm to gradually find all distinguishing vectors. Consider Figure 6. We first isolate the faulty regions in the old implementation and the new implementation. We then connect both faulty regions together by an XOR gate, *xor* in Figure 6(b). Then, we set a stuck-at-0 at the output of *xor* and use an ATPG algorithm to find a test vector for the stuck-at-0 fault. Any test vector for the stuck-at-0 fault is a differentiating vector for the pseudo output i_4. Note that our goal is to find all differentiating vectors. Figure 6(c) shows a vector $v_1 = (i_1, d, e) = (0,1,1)$ is a differentiating vector. Then, we construct a product term Boolean function by performing the AND function of variables in the differentiating input vector. And feed the product term Boolean function into XOR gate, *xor*. In the example, after $v_1 = (i_1, d, e) = (0,1,1)$ is found, we produce the Boolean function of $i_1'de$ and connect the function to the input of XOR gate, *xor* in Figure 6(d). The reason to insert a product term function into the XOR gate is to prevent our ATPG algorithm from finding the same differentiating vector in the next iteration. After the insertion, we then perform again the ATPG algorithm to find the stuck-at-0 fault for output *xor*. The algorithm will find another differentiating vector which is different from the previous one. In the same example, we can obtain another differentiating vector $v_2 = (i_1=0, d=0, e=1)$. This process is

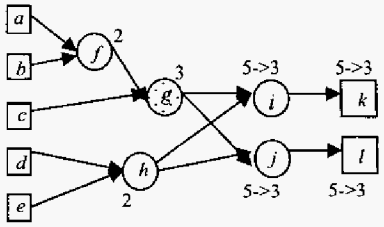

Figure 4 reduction of dependency number by choosing g as pseudo input/output

continued until no differentiating vector is found. Note that there are a lot of equivalent nodes between the old and new implementation. To reduce the ATPG complexity, we can share the internal signals between the old implementation and the new implementation. For example in Figure 6(b), the function of node i_2 in both the original and new implementation is identical, so we can share the function between two implementations. The simplified circuit is shown in Figure 6(e).

6. Design Consideration for Input/Output Select Logic and One Mask Programmable Design Style

Note that the size of some programmable logic devices such as PLA is directly related to the number of their inputs. The use of the input-select logic allows us to have a reasonable size of inputs of the rectification module selected from a much larger set of possible pseudo/primary inputs. The objective of an input select logic is to reduce the size of the rectification module while keeping the flexibility of using many possible inputs. Because both input-select and output-select logics have the similar functionality, we only discuss the input-select logic in this section. The functionality of the input-select logic is to choose a subset from a large set of signals. Suppose we would like to design an input-select logic which can select any 5 signals $\{i_1, i_2, i_3, i_4, i_5\}$ from 10 signals $\{j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, j_{10}\}$. A simple solution is to have full programmability which allows any i signal to be able to connect to any j signal. However, it is not necessary to have full programmability. In Figure 5, signal i_1 is not required to connect to signal j_{10}. In fact, i_1 needs only has possible connection with j_1, j_2, j_3, j_4, j_5 and j_6. On the other hand, i_2 needs only has possible connection with j_2, j_3, j_4, j_5, j_6 and j_7. In this way, we can reduce the design area of input-select and output-select logics by reducing unnecessary programmability.

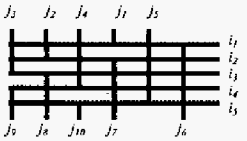

Figure 5 one choice of the select logic

7. Experimental Results

We have performed a set of experiments to demonstrate the usefulness of our framework. In our experiments, an engineering change problem is created by inserting an error to a circuit. An error is either by inverting a node's output or inputs, modifying some cubes of a node, or changing a node's gate type. In Table 1, column one, two and three give the name, the gate count, and the number of primary inputs/outputs of a circuit, respectively. Column four titled "#Pseudo Primary Inputs/Outputs" reports the number of pseudo primary inputs/outputs generated under four different value of predetermined number k (k=10, 15, 20, and 25.) Column five presents, in a sum-of-product (SOP) form, the number of literals required for the rectification logic.

Column six shows the PLA size needed to implement the rectification logic. Column seven presents the run time of generating the results in column four and five. Finally, column eight demonstrates, in a multi-level form of fac, the number of literals required for the rectification logic. For example, the first row in Table 1 shows that circuit C432 has 160 gates, 36 primary inputs, and 7 primary outputs. When the k=10, our framework locates 47 pseudo primary inputs/outputs in the circuit. Further, given a random error, our framework derives a rectification logic involving 4 literals in a SOP form and implemented by a PLA block whose size is at least 29.9 μm^2. In addition, the rectification logic involves 4 literals in a multi-level form. Also, in the table, the symbol NA means that a result cannot be obtained due to the out of memory problem.

8. Conclusions

We have implemented a framework to perform the design for engineering change by using the rectification module approach targeting at reducing the mask set cost and turn around time. The framework includes both the design of a one-mask programmable PLA as the rectification module and the design automation software to find the rectification logic. The design automation software selects a set of pseudo inputs/outputs and finds the rectification logic for a given engineering change problem. The experimental results from a set of MCNC benchmark circuits show that by properly selecting internal nodes, the rectification logic can be greatly reduced.

References

1. S. C. Chang and J. C. Rau, "A Timing-Driven Pseudo-Exhaustive Testing of VLSI Circuits," IEEE International Symposium on Circuits and Systems, May, 2000.
2. P. Y. Chung, Y. M. Wang, and I. N., Hajj, "Logic Design Error Diagnosis and Correction," IEEE Transactions on Very Large Scale Integration (VLSI) System., September 1994.
3. D. Brand, A. Drumm, S. Kundu, and P. Narrain, "Incremental Synthesis," in Proc. Int. Conf. Computer-Aided Design, Nov. 1994, pp. 14-18.
4. C.-C Lin, K.-C. Chen, and M. Marek-Sadowska, "Logic Synthesis for Engineering Change," IEEE Transactions on Computer-Aided Design of Integrated Circuits And Systems, March 1999.
5. S.-Y Huang, K.-C Chen, and K.-T Cheng, "AutoFix: A Hybrid Tool for Automatic Logic Rectification," IEEE Transactions on Computer-Aided Design of Integrated Circuits And Systems, September 1999.
6. R. Srinivasan, S. K. Gupta and M. A. Breuer, "An Efficient partitioning Strategy for Pseudo-Exhaustive Testing," *Proc. Design Automation Conf.*, 1993, pp.242-248.
7. Y. Watanabe and R. K. Brayton, "Incremental Synthesis for Engineering Changes," in Proc. Int. Conf. Computer-Aided Design, Nov. 1991, pp. 130-133.

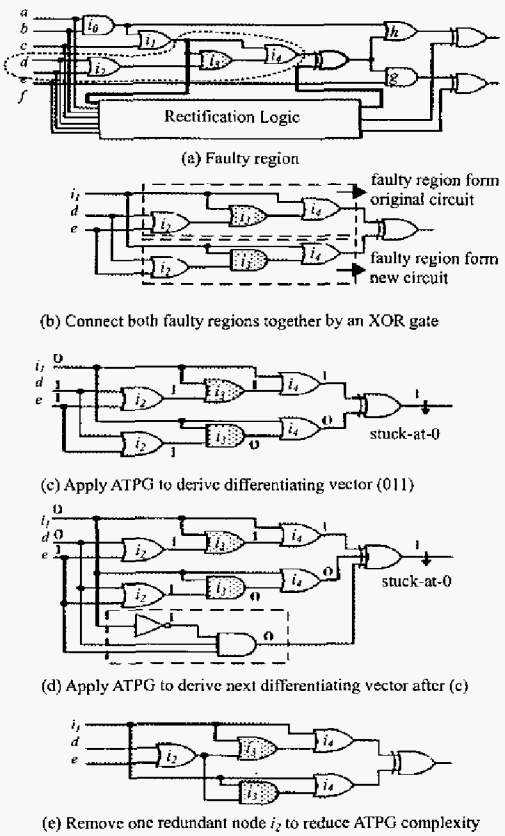

Figure 6 Obtain the rectification logic

Table 1 Experimental results with various dependency numbers and an error injection

circuit	#gates	#PI/PO	#Pseudo Primary Input/Output				#literals of RM in SOP format				PLA Sizes of RM (um^2)				CPU Time				#literals of RM in multi-level form			
			K=10	K=15	K=20	K=25	K=10	K=15	K=20	K=25	K=10	K=15	K=20	K=25	K=10	K=15	K=20	K=25	K=10	K=15	K=20	K=25
C432	160	36/7	47	32	21	18	4	19	367	1436	30	189	4140	12474	0.64	1.07	878	5008	4	10	96	227
k2	745	45/45	564	330	120	49	4	38	41	41	48	201	212	212	22.39	15.13	16.88	20.14	4	15	17	17
vda	357	60/26	274	38	0	0	20	20	20	20	125	125	125	125	6.22	5.67	5.13	5.15	19	19	19	19
too_large	7613	38/3	5700	1966	458	120	28	62	64	64	142	294	323	323	4859.20	1133.07	133.71	34.09	13	18	20	20
C1908	880	33/25	37	20	15	13	7	28	51	762	39	151	266	4403	5.83	6.35	7.27	17.06	7	13	21	64
C3540	1667	50/22	286	102	72	51	13	12	12	13	83	83	83	83	1016	346.2	247.5	194.7	10	9	9	10
i9	335	88/63	7	0	0	0	10	63	63	63	104	502	502	502	13.97	11.65	11.56	11.40	7	29	29	29
dalu	1131	75/16	107	85	56	28	10	118	174	174	72	744	785	785	13.61	16.79	24.66	39.64	8	47	56	56
frg2	522	143/139	47	8	4	0	9	57	133	161	72	589	2059	2216	19.14	26.44	4191	15828	7	15	24	27
x3	332	135/99	36	3	1	0	24	24	36	36	166	166	201	201	7.89	8.84	9.32	28.59	9	9	12	12
i8	1085	133/81	344	63	0	0	27	29	397	397	189	251	3447	3447	49.02	31.17	55.39	55.38	14	17	127	127
i10	1652	257/224	207	97	38	22	5	25	39	39	33	222	212	212	159.6	125.5	104.6	100.5	5	9	15	15
term1	147	34/10	39	14	0	0	16	70	102	102	89	402	494	494	0.48	0.69	1.22	1.22	11	17	21	21
apex6	550	135/99	24	7	1	0	2	2	576	576	24	24	3481	3481	10.04	9.65	17.37	17.22	2	2	144	144
t481	2072	16/1	640	61	0	0	6	8	36	36	36	42	189	189	54.60	6.86	465.4	465.4	6	8	14	14