Efficient Boolean Division and Substitution

Shih-Chieh Chang
Dept. of CS and IE
National Chung Cheng University
Chia-Yi, TAIWAN, ROC

David Ihsin Cheng
Ultima Interconnect Tech.
Sunnyvale, CA 94086

Abstract

Boolean division, and hence Boolean substitution, produces better result than algebraic division and substitution. However, due to the lack of an efficient Boolean division algorithm, Boolean substitution has rarely been used. We present an efficient Boolean division and substitution algorithm. Our technique is based on the philosophy of redundancy addition and removal. By adding multiple wires/gates in a specialized way, we tailor the philosophy onto the Boolean division and substitution problem. From the viewpoint of traditional division/substitution, our algorithm can perform substitution not only in sum-of-product form for but also in product-of-sum form. Our algorithm can also naturally take all types of don't cares into consideration. As far as substitution is concerned, we also discuss the case where we are allowed to decompose not only the dividend but also the divisor. Experiments are presented and the result is promising.

1 Introduction

In multi-level logic synthesis, an important step in minimizing the area of a circuit is *substitution* [4] (or *resubstitution* [10]). Substitution refers to the step where a function is simplified in complexity by using an additional input that was not previously in the function's immediate fanins. Substitution can reduce the complexity of a function because part of the function is replaced by the additional input that represents some existing function in the circuit. The expression of the existing function is therefore shared and reused. To perform substitution, the concept of *division* plays a major role. Given two Boolean functions F and D, if we can express F in the form $F = Q \cdot D + R$, where $\cdot$ and $+$ respectively represent the Boolean AND and Boolean OR operators, then we say that F can be *divided* by D and that functions Q and R are, respectively, the *quotient* and the *remainder*.

Substitution can be algebraic or Boolean, depending on if the underlining division is algebraic or Boolean. In algebraic division [4], logic expressions are treated as algebraic polynomials, with some restrictions placed on the manipulations of the polynomials. In particular, the product of two functions $F \cdot G$ is algebraic only if no variable appears in both F and G. As a consequence of the restriction, certain Boolean identities such as $x \cdot \overline{x} = 0$ and $x \cdot x = x$ do not exist. As an example, given $F = \overline{a}b + bc + ac$ and divisor $D = a + b$, through algebraic division we obtain $F = c(a + b) + \overline{a}b$. Through Boolean division, which can exploit all the properties in Boolean algebra [1], we obtain $F = (\overline{a} + c)(a + b)$. Assuming a node d with function $a + b$ exists in the circuit, with algebraic substitution we then have $F = cd + \overline{a}b$, while with Boolean substitution we have $F = (\overline{a} + c)d$. In this example, function F has 5 literals[1] before substitution. Algebraic substitution reduces the number of literals to 4, while Boolean substitution reduces it to 3. Boolean division, and hence Boolean substitution, in theory produces better results. However, there does not exist a general and efficient Boolean division algorithm. In terms of the above example, this means that the best result of reducing F to 3 literals is very difficult to achieve.

In this paper, we first present a new technique to perform Boolean division. Our technique is based on the concept of *redundancy addition and removal* (RAR) discussed in [2][3][5][8]. The basic philosophy of the RAR technique is to first add some redundancy and then remove other redundancies elsewhere, with the goal that the removed ones reduce the circuit size more than the added one. With a fixed setup that is specially configured, we tailor the RAR philosophy onto the Boolean division problem. Unlike traditional RAR techniques, which require redundancy checking on the potential wire to be added, our algorithm is tailored in a way that we know a priori that our interested potential wire is redundant. Also, although quite effective on adding one redundancy and then removing other redundancies, the traditional RAR techniques have little success on trying to add multiple wires/gates. In our algorithm, the traditional RAR philosophy is tailored to add multiple wires/gates in a specific way particularly for the Boolean division problem.

As far as substitution is concerned, knowing how to perform division is only the first step. The second step is to choose potential divisors. Traditionally, substitution on a function F is done by going through the existing nodes in the circuit and treating each of them

[1] In *factored form* [4], as opposed to sum-of-product form.

as a potential divisor of F. Division is tried on each potential divisor and substitution is carried out when the trial is favorable. Since it is up to the underlining division algorithm to conclude if a divisor is good or not, the algorithm may miss some "good" divisors. In the example mentioned earlier, let us say the node with function $a+b$ does not exist and, instead, a node with function $D = a+b+x$ exists. Since function F does not depend on variable x, a traditional division algorithm would quickly conclude that the quotient of function F divided by D is 0, and therefore no substitution would occur. However, if we slightly change the circuit structure by decomposing $a+b+x$ to two nodes $n_1 = a+b$ and $n_2 = n_1+x$, function F can then be substituted with node n_1. We will use the term *basic division* to refer to the scenario where the given divisor is not allowed to be decomposed, and the term *extended division* for the scenario where the divisor is allowed to be decomposed, certainly with some purpose in mind. In the above example where function $F = \overline{a}b+bc+ac$ is divided by $D = a+b+x$, we would say that under basic division the quotient is 0. For the same F and D we would also say that under extended division the subexpression $a+b$ can be extracted out as a new divisor, and with the new divisor $a+b$ the quotient is $\overline{a}+c$. From this viewpoint, all the traditional division algorithms perform only basic division, while our algorithm presented in this paper performs extended division.

Traditional substitution approaches operate on each node's internal sum-of-product data structure, and hence can only perform substitution/division in the sum-of-product form. In contrast, our algorithm operates on circuit structure directly. Given an initial circuit, the first step of our algorithm is to decompose each node's internal sum-of-product form into a two-level AND and OR gates. The circuit then, in general, has a level of AND gates, followed by a level of OR gates, and so on. As a result, in addition to the traditional sum-of-product type of substitution, our algorithm can also perform substitution in the flavor of product-of-sum form. In other words, in two-level form, whether the dividend/divisor are a bunch of ANDs followed by an OR, or a bunch of ORs followed by an AND are completely symmetric to us. For example, let $F = (\overline{a}+b)(b+c)(a+c)$ and $D = (a)(b)$ be existing nodes. With our algorithm we can quickly substitute D into F and obtain $F = D+\overline{a}c$, i.e., $F = ab+\overline{a}c$. Performing substitution in such a manner is completely not possible in the traditional approaches because of the strong attachment to the underlining sum-of-product expression, while in our technique performing substitution through sum-of-product form or product-of-sum form are basically the same.

2 Redundancy addition and removal

The most related work to our Boolean substitution algorithm is the technique of *redundancy addition and removal* (RAR). Here we provide a detailed review. In [2][3][5][8], the technique of RAR is proposed and applied to general multi-level logic optimization. The basic philosophy in RAR is to add some redundancy first and then try to remove other redundancies elsewhere, with the goal that the removed ones reduce the circuit size more than the added one. We review the technique with an example circuit. Fig. 1(a), without the dotted wire, shows an irredundant circuit. The dotted wire $g5 \rightarrow g9$ is a redundant wire, i.e., adding the wire does not change the circuit's functionality. However, once this wire is added, the two thick wires, $g1 \rightarrow g4$ and $g6 \rightarrow g7$, become redundant. In this case, we can remove these two redundant wires without changing the circuit's functionality. After removing these two wires, we then have the circuit shown in Fig. 1(b), which is smaller in size.

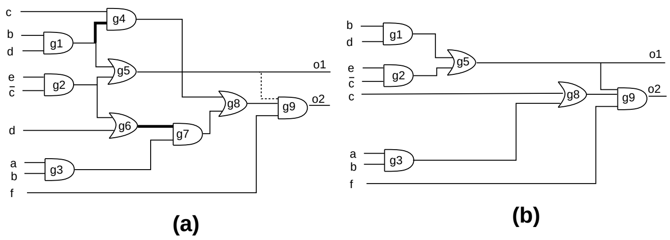

Figure 1: The RAR technique

In general, the RAR technique first decides, based on some cost function, some existing irredundant wire that is the target to be removed. Then the technique searches for some non-existing wire, sometimes called a candidate connection, that once added can remove the target wire. Finally the technique checks if the candidate connection is redundant, i.e., if adding the non-existing wire preserves the circuit's functionality. Only when the candidate connection is verified as redundant, we can then add the connection and further remove the target wire. Note that most of the RAR techniques only try to incrementally add one wire at a time. Due to a high search space, efforts that try to add multiple wires/gates and remove even more wires/gates have only little success (e.g.: [2]).

3 Basic division

Given a function F and a divisor D, we use the term *basic division* to refer to the scenario where the divisor is not allowed to be decomposed, and use the term *extended division* to refer to the scenario where the divisor can be freely decomposed, with some optimization goal in mind. In this section we focus on basic division.

3.1 SOS and POS of a function

We first need some definitions. A *product term*, or *cube*, is a set of literals AND'ed together. A *sum term* is a set of literals OR'ed together. A function f_1 *contains* a function f_2 if the on-set of f_1 contains the on-set of f_2. As an example, function (cube) a contains function (cube) ab; function (sum term) $a+b$ contains function (sum term) a. Furthermore, we define SOS and POS of a function as follows:

- **SOS:** Given a function F in two-level sum-of-product form, we say a function G, also in sum-of-product form, is a *sum-of-subproduct*, or SOS, of F if every cube in F is contained by at least one cube in G.

- **POS:** Given a function F in two-level product-of-sum form, we say a function G, also in product-of-sum form, is a *product-of-subsum*, or POS, of F if every sum term in F contains at least one sum term in G.

For example, $D = a+b$ is a SOS of $F = \overline{a}b+bc+ac$ because every cube in F is contained by either cube a or cube b in D. For another example, $D' = a+b+x$ is also a SOS of the above F, since adding more cubes to D does not change the original containment relationship in F. On the other hand, function $E = ab+c$ is not a SOS of F, since cube $\overline{a}b$ is not contained in any cube in function E.

On the POS side, for example, $D = (a)(b)$ is a POS of $F = (\overline{a}+b)(b+c)(a+c)$ because every sum term in F contains either sum term a or sum term b in D. For another example, function $D' = (a)(b)(x+y)$ is also a POS of the above F, since adding more sum terms to D does not change the original containment relationship in F. On the other hand, function $E = (a)(b+c)$ is not a POS of F, since sum term $\overline{a}+b$ does not contain any sum term in function E.

The concepts of SOS and POS play a central role in our algorithm, and we now look at some of their simple properties.

Lemma 1 *Let function G be a SOS of function F. Then $F = F \cdot G$.*

Lemma 2 *Let function G be a POS of function F. Then $F = F + G$.*

These two lemmas establish the ground where we can tailor the technique of redundancy addition and removal (RAR) onto our substitution problem. To illustrate the concept, we take the example of $F = \overline{a}b + bc + ac$ and $D = a + b$ from Section 1. Since D is a SOS of F, by Lemma 1, the new function $F_{new} = (a+b)(\overline{a}b + bc + ac)$ must be equivalent to the original function F. From the RAR viewpoint, we have successfully "added" a redundancy into the circuit. Focusing on the original F part inside F_{new}, we then try to remove as many redundancies as possible, and can quickly arrive $F = (a+b)(\overline{a}+c)$. Symmetric to the SOS case, we can perform similar operations on POS. Let $F = (\overline{a}+b)(b+c)(a+c)$ and $D = (a)(b)$. Since D is a POS of F, by Lemma 2, the new function $F_{new} = (a)(b) + (\overline{a}+b)(b+c)(a+c)$ must be equivalent to the original function F. Focusing on removing redundancies from the original F part inside F_{new}, we then quickly have $F = ab + \overline{a}c$.

3.2 Performing basic division

Given a function F and a divisor D, in this section we present an algorithm that performs basic Boolean division, i.e., $F = Q \cdot D + R$. The best way to explain our algorithm is to discuss it with an example. Fig. 2(a) shows two nodes, which correspond to $F = \overline{a}b + bc + ac + y$ and $D = a + b$. Since our central idea is based on the SOS concept, the first step to

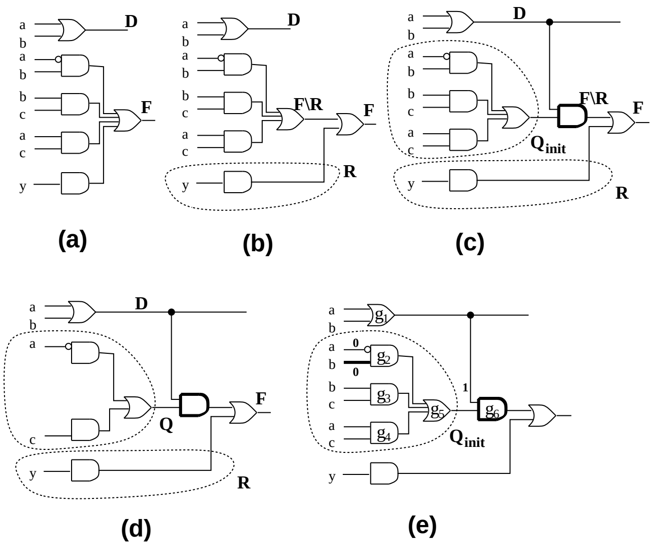

Figure 2: Basic division

perform F divided by D is to take out from F all the cubes that are not contained by any cube in D, and such cubes will be our final remainder term R. Among the four cubes in F, y is the only such cube since $y \not\subseteq a$ and $y \not\subseteq b$. Fig. 2(b) shows the circuit structure after we form the remainder y, where we use dotted circle R to indicate the remainder region and $F\backslash R$ to denote the resulting function with cube y taken out from F. Since every cube in $F\backslash R$ is now contained by at least one cube in D, D is a SOS of $F\backslash R$. By Lemma 1, $F\backslash R$ would stay unchanged if AND'ed with D. This fact is shown in Fig. 2(c) with an extra bold AND gate and the shift of $F\backslash R$ from before this AND gate to after this AND gate. From the viewpoint of the RAR technique, we have successfully added a redundancy and the circuit still has the same functionality. Now the region marked by the circle Q_{init} is highly redundant. The final step is to perform redundancy removal on the Q_{init} region and we reach the final result shown in Fig. 2(d), which is of the form $F = Q \cdot D + R$. To show how redundancy removal is done, we duplicate the circuit snapshot shown in Fig. 2(c) to Fig. 2(e) and remark some nodes. Let us illustrate how wire $b \rightarrow g_2$, the thick wire in Fig. 2(e), is detected as a redundant wire. For wire $b \rightarrow g_2$ stuck-at-1 fault to be testable, b must be 0 to activate the fault. For the fault effect to propagate through gate g_2, a must be 0. For the fault effect to propagate through gate g_6, g_1 must be 1. Since $a = 0$ and $b = 0$ implies $a + b = 0$, gate g_1 must be 0, which is a conflict. A conflict during the implication process means the fault $b \rightarrow g_2$ stuck-at-1 is untestable, and therefore wire $b \rightarrow g_2$ can be replaced by a constant 1. Our basic division algorithm works as illustrated by the above example. In summary, our algorithm consists of three steps. The first step of our algorithm is to decompose the dividend F so that the cubes that make the divisor D not a SOS of F form the remainder R. The second step is to AND D with $F\backslash R$, which does not change the functionality of $F\backslash R$ by Lemma 1. The third step is to remove all the redundancies inside the $F \backslash R$ region.

Note that it is the redundancy addition and removal steps that make our technique Boolean. Comparing to the traditional RAR techniques, however, a major difference lies on the fact that we know *a priori* that the added wires/gates are redundant because of the SOS property in Lemma 1. In other words, unlike the traditional RAR techniques, we do not need to check if the added wire/gate are redundant or not. Furthermore, as mentioned in Section 2, there has been little success in works trying to generalize the RAR technique to adding multiple wires/gates. What our algorithm does is essentially a tailored version of the RAR philosophy onto the substitution problem, with a fixed configuration of multiple wires/gates addition. Also note that since the added wires/gates are known to be redundant a priori, the most time-consuming step in our algorithm is only on the redundancy removal step. With different implication methods (e.g.:[6][7][9]) we can actually adjust the tradeoff between the run time and the quality of result. For example, we can limit our implication process only inside a small region, the $F \backslash R$ region plus the D region. As far as substitution is concerned, most of the reconvergences and implication conflicts would occur in this small region. Limiting the implication process inside this small region would greatly reduce the time required as opposed to a traditional redundancy removal process. On the other hand, we can certainly spend more time to perform implications to gates outside this small region, and thereby naturally incorporating any external or internal don't cares into consideration. In the extreme case, we can even adopt some quite exhaustive implication technique such as recursive learning [7] to incorporate a large amount of internal don't cares. We do not discuss the details here but simply point out the existence of such a flexibility on various implication algorithms. Finally, as can be seen from the above example, our algorithm operates on circuit structure directly, rather than manipulating expressions like traditional approaches. As mentioned earlier, we therefore are not limited to doing substitution only in terms of the traditional sum-of-product viewpoint. With the POS concept, we can also perform substitution on two functions when they are both in the product-of-sum form. Instead of using the SOS concept and Lemma 1, we can use the POS concept and Lemma 2, and the same philosophy as illustrated above would apply directly. As a simple example, imagine a circuit that is identical to the one shown in Fig. 2 with all the AND gates changed to OR gates and vice versa. With our algorithm it is as easy as was illustrated in this section, while in a traditional substitution technique all the sum-of-product expressions form a complete new problem whose result is difficult to predict. Since conceptually SOS and POS are symmetric, throughout the remaining of this paper we do not go into the details of the case for POS.

4 Extended division

The previous section presented our algorithm that performs basic division, where a divisor is not allowed to be decomposed. Given a function F and a divisor D, under basic division we seek to reexpress F as $F = Q \cdot D + R$. This means we are only allowed to decompose F but not on D. In this section we present an algorithm that performs what we call extended division. Given a function F and a divisor D, under extended division we are allowed to decompose not only F but also D, with the purpose of minimizing the number of literals in substitution. In essence, we first want to separate the cubes in D into two groups, the *core divisor* D_C and the *remaining divisor* D_R. Once this separation is determined, we decompose the original divisor D into two nodes such that $D = D_C + D_R$. Decomposing into a new node for the core divisor D_C means that D_C, a subexpression that was originally embedded in the given divisor D, is now exposed and can be used for substitution. We then apply our basic division algorithm in the previous section on function F and core divisor D_C to obtain the result. For example, given function $F = \overline{a}b + bc + ac + y$ and divisor $D = a + b + x$, we decompose the divisor D into the core divisor $D_C = a + b$ and the remaining divisor $D_R = x$. Applying our basic division algorithm on F and D_C, we then obtain the same result as illustrated in the previous section. It should be clear that the most important thing here is to intelligently determine the core divisor D_C, since once D_C is determined an extended division reduces to a basic division.

Recall that during our basic division algorithm, it is the step of redundancy removal that really performs the minimization process. Looking back in Fig. 2(c), whenever we remove a wire from the cubes in the Q_{init} region, we effectively reduce a literal in the final quotient. What we would like to have is a core divisor that is able to remove the most wires. To determine the core divisor D_C with a given function F and a given divisor D, our basic idea is to have each wire in the cubes of F "vote" for a candidate core divisor. For each wire w in the cubes of F, we perform implications to see which cubes in divisor D are able to remove wire w. For example, let function $F = abd + \overline{a}c + cd + ae$ and divisor $D = ab + c + f + bd + e$, whose circuit structure is shown in Fig. 3(a). In Fig. 3(a), we name divisor D's five cubes y_1, y_2, y_3, y_4 and y_5; we also name function F's four cubes x_1, x_2, x_3 and x_4, which are respectively driven by gates g_1, g_2, g_3 and g_4. Consider wire $a \rightarrow g_1$ stuck-at-1 fault. We have the following implications.

$a = 0$ (activate fault)	$\Longrightarrow$	$y_1 = 0$
$b = 1$ & $d = 1$ (allow fault thru g_1)	$\Longrightarrow$	$y_4 = 1$
$x_2 = 0$ (allow fault thru g_5) & $a = 0$	$\Longrightarrow$	$c = 0$
$c = 0$	$\Longrightarrow$	$y_2 = 0$

Assume for now that we somehow have determined a core divisor D_C. This core divisor, in our specialized configuration for basic division, feeds into a gate similar to the bold AND gate g_6 in Fig. 2(e) of the previous section. This means that if we want any fault effect in the Q_{init} region to propagate through the bold AND gate, this core divisor D_C must have a value 1 during the fault's implication process. In the case of extended division, if the core divisor that we eventually deter-

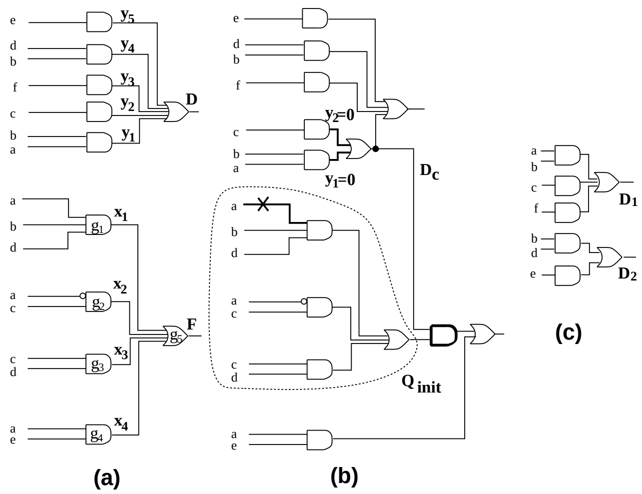

Figure 3: Extended division

wire	$y_i = 0$
$a \to g_1$	y_1, y_2
$b \to g_1$	y_1, y_2, y_4, y_5
$d \to g_1$	y_4, y_5
$a \to g_2$	
$c \to g_2$	y_1, y_2
$c \to g_3$	y_1, y_2
$d \to g_3$	y_4, y_5
$a \to g_4$	y_1
$e \to g_4$	y_4, y_5

(a) initial

wire	$y_i = 0$
$a \to g_1$	y_1, y_2
$b \to g_1$	y_1, y_2, y_4, y_5
$d \to g_1$	y_4, y_5
$a \to g_2$	
$c \to g_2$	y_1, y_2
$c \to g_3$	y_1, y_2
$d \to g_3$	
$a \to g_4$	
$e \to g_4$	y_4, y_5

(b) final

Table 1: Vote table

mine has implication value 0 for a particular fault, the fault must be untestable because a conflict will occur with the required assignment of 1 mentioned above. We illustrate this point by continuing the example for wire $a \to g_1$ stuck-at-1 fault. We focus on the results that appear on the y_i's side whose implication values are 0. In this case, we have $y_1 = 0$ and $y_2 = 0$. Assuming we eventually choose $y_1 + y_2$ as our final core divisor, i.e., $D_C = y_1 + y_2 = ab + c$, then our basic division algorithm in the previous section would change the circuit structure to the one shown in Fig. 3(b), where $D_C = ab+c$ is connected to the bold AND gate. Following the basic division algorithm in the previous section, we would try to remove as many wires as possible in the Q_{init} region. When we again perform implications for the fault $a \to g_1$ stuck-at-1 fault, shown with a cross in Fig. 3(b), we know that y_1 and y_2, and hence D_C, all have implication value 0. This creates a conflict because, as stated earlier, for the fault effect of $a \to g_1$ stuck-at-1 to propagate through the bold AND gate, D_C must be assigned 1. In other words, if we do choose $y_1 + y_2$ as our core divisor, we expect wire $a \to g_1$ to be removed in the subsequent basic division. Now, in determining the core divisor, different wires have different implication values on the y_i's side in Fig. 3(a). In some sense, this means that each wire "votes" for a candidate core divisor. In the above example, wire $a \to g_1$ votes for candidate core divisor $y_1 + y_2$. This should become clear if we look at the complete situation after each wire performs implications on the example circuit shown in Fig. 3(a). Table 1(a) lists all the y_i's that have implication value 0 for each wire.

We explain the interpretation of Table 1(a) by examples. The meaning of the second row is that we expect wire $b \to g_1$ to be removed if we choose $y_1 + y_2 + y_4 + y_5$ as the core divisor. For simplicity, we also say that wire $b \to g_1$ *votes* for candidate core divisor $y_1 + y_2 + y_4 + y_5$. Similarly, the meaning of the fourth row is that we do not expect wire $a \to g_2$ to be removed, regardless of whatever core divisor we choose, and hence has no candidate core divisor. The remaining entries of Table 1(a) can be interpreted in a similar way.

The above voting scheme demonstrates our criteria for choosing a good core divisor. From the RAR technique's viewpoint, however, one more thing we need to make sure is that a candidate core divisor is indeed a redundant wire which we can eventually "add" to the circuit. This is done by checking if the candidate core divisor voted by a wire w is a SOS of the cube that is connected to wire w. For example, from the first entry in Table 1(a), the candidate core divisor of wire $a \to g_1$ is $y_1 + y_2 = ab + c$. The cube that is connected to wire $a \to g_1$ is $x_1 = abd$. Since the candidate core divisor $ab + c$ is a SOS of cube abd, we know eventually if we add core divisor $ab + c$ into the circuit, the added wire will be a redundant wire and therefore the circuit functionality would not change. In Table 1(a), the only candidate core divisors that do not hold for this condition are wires $d \to g_3$ and $a \to g_4$. The candidate core divisor for wire $d \to g_3$ is $y_4 + y_5 = bd + e$, which is not a SOS of the corresponding cube $x_3 = cd$. On the case of wire $a \to g_4$, candidate core divisor $y_1 = ab$ is not a SOS of the corresponding cube $x_4 = ae$. We therefore need to delete these two entries in Table 1(a) and we have our final *vote table*, shown in Table 1(b). To finalize the choice of the core divisor, various heuristics can be used. We reduce the above choice problem to a maximal clique problem in graph theory. Due to space limit, we omit the details here.

Applying our extended division algorithm to the substitution problem, we want to point out that we can actually do more than what the above discussion shows. In the above formulation, we focus only on one existing node D. In the case of substitution, we actually have freedom to select our core divisor from among many circuit nodes. As an example of how this generalization works, imagine the given divisor in the above example, $D = ab + c + f + bd + e$, does not exist in our circuit and instead, two nodes $D_1 = ab + c + f$ and $D_2 = bd + e$ exist, as shown in Fig. 3(c). When function $F = abd + \overline{a}c + cd + ae$ is given and we want to search for a good divisor between D_1 and D_2 with extended division, we can temporarily pretend that all the five cubes are from the same node, and therefore the flow is identical to the example shown in this section. Each wire in the cubes of D_1 and D_2 votes for a candidate core divisor and we have an identical vote

bench	init	sis		basic		ext.		ext.+GDC	
	lit.	lit.	cpu	lit.	cpu	lit.	cpu	lit.	cpu
C1355	560	560	1.35	560	1.96	558	1.98	558	7.59
C1908	679	632	2.61	598	1.65	598	1.70	596	3.32
C2670	939	849	3.65	840	2.63	831	2.80	828	10.03
C3540	1635	1618	7.86	1300	17.39	1345	16.82	1332	315.26
C499	560	560	0.82	560	1.94	558	1.96	558	6.50
C5315	1961	1927	8.02	1870	5.94	1864	6.44	1856	24.11
C6288	4212	4212	37.19	3747	20.64	3747	21.10	3747	21.71
C7552	2627	2534	15.94	2382	13.13	2390	14.38	2367	113.62
alu4	864	857	81.63	821	16.57	833	15.70	829	328.24
apex6	817	817	1.46	803	3.57	802	3.79	800	24.81
dalu	1809	1765	17.07	1671	37.09	1609	38.07	1587	895.06
des	4920	4918	80.63	4728	110.30	4734	115.31	4733	3325.34
frg2	1177	1135	6.82	1066	14.63	1071	15.04	1058	253.36
i10	2772	2757	13.23	2567	44.18	2558	46.17	2518	608.61
i8	1491	1491	14.04	1211	34.82	1221	39.21	1216	850.84
i9	749	749	3.07	730	10.77	730	14.16	687	413.90
rot	758	763	1.73	710	1.95	706	1.90	703	5.00
t481	1395	1313	58.80	1114	17.37	1111	15.66	1108	195.26
term1	357	349	2.93	319	1.40	323	1.31	309	5.45
x3	975	966	3.36	932	4.77	946	5.01	930	27.62
x4	522	492	1.56	488	2.21	469	2.62	457	16.81
s1196	626	624	3.10	577	5.80	573	5.89	569	83.57
s13207	3569	3552	28.84	2894	37.57	2836	36.54	2828	931.91
s1423	652	652	2.58	621	1.87	624	1.88	621	6.55
s1488	762	758	4.97	722	18.48	714	17.25	710	467.52
s1494	756	754	5.05	724	16.97	713	16.09	709	453.01
s15850	4306	4269	29.81	3789	34.16	3760	35.71	3751	605.06
total	47134	46547	503	42642	511	42496	526	42200	10347
%		1.2%		9.5%		9.8%		10.5%	

Table 2: Experimental results

table as shown in Table 1(b). The only slight modification we need is in the final maximal clique formulation, where we need to model the fact that some cubes in the second column of Table 1(b) originally come from a different node. Note that, as is also the case for basic division, we can perform extended division in terms of sum-of-product form as well as product-of-sum form. Instead of focusing on the cubes that have implication value 0, we would then focus on the sum terms that have implication value 1. The rest of the algorithm applies similarly.

5 Experimental results

We have implemented our algorithm and applied it to the substitution problem. Our implementation has three configurations: basic division, extended division without considering global internal don't cares, and extended division with global internal don't cares taken into consideration. We performed experiments on MCNC and ISCAS benchmarks within SIS [10] environment. We first run the script

eliminate 0; simplify -d; gcx;

on each benchmark to obtain the initial circuit. We then compare our algorithm with the algebraic resubstitution "resub -d" in SIS. Table 2 shows the comparison between SIS and our results. The first column shows the name of the circuit. The second column shows the initial literal count. The columns labeled "sis" is the result of running "resub -d" command, with subcolumns "lit." and "cpu" reporting the number of literals and CPU time, respectively. All literal counts are in factor form. Take the circuit *C2670* as an example, after running the above script, initially the circuit had 939 literals, shown in the second column. After running "resub -d" the literal count was reduced to 849 literals. With our basic division it is reduced to 840 (Column "basic"). The extended division reduced it to 831 (Column "ext."). The extended division with global don't cares taken into account brought it down to 828 (Column "ext.+GDC"). The last two rows show the summation of each column and the percentage of improvement compared to the initial literal count. As the table indicates, our division algorithms outperformed the traditional division and substitution. In terms of run time, "resub -d", the basic division, and the extended division without global don't cares spent similar CPU times. The extended division with global don't cares spent the most time.

6 Conclusion

In this paper we first presented an efficient algorithm, based on the philosophy of redundancy addition and removal (RAR), for performing Boolean division. With the concept of SOS and POS, we tailored the RAR philosophy to the Boolean division problem. The tailoring enables us to add multiple wires/gates in a specialized configuration and remove more wires/gates. Applying our Boolean division algorithm, our algorithm can perform substitution not only in the traditional sum-of-product form, but also in product-of-sum form. We then generalize our basic division to what we call extended division. Extended division allows us to decompose not only on the dividend but also on the divisor. Furthermore, our technique is able to naturally incorporate all types of don't cares into consideration. We also presented some experimental results to verify the effectiveness of our algorithm.

References

[1] R.K. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions," *Proc. ISCAS,* pp. 49-54, 1982.

[2] S.C.Chang and M.Marek-Sadowska, "Perturb and simplify: multi-level Boolean network optimizer," *Proc. ICCAD,* pp. 2-6, Nov. 1994.

[3] S.C. Chang, L. VanGinneken, and M. Marek-Sadowska, "Fast Boolean Optimization by Rewiring," *Proc. ICCAD,* pp. 262-269, 1996.

[4] G. De Micheli, "Synthesis and Optimization of Digital Circuits," *Mc Graw Hill Text,* 1994.

[5] L.A.Entrena and K.T.Cheng, "Combination and sequential logic optimization by redundancy addition and removal," *IEEE TCAD,* Vol. 14, no. 7, pp. 909-916, Jul. 1995.

[6] T. Kirkand and M.R. Mercer, "A Topological Search Algorithm For ATPG," *Proc. DAC,* pp. 502-508, 1987.

[7] W. Kunz and D.K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," *Proc. ITC,* pp. 816-825, 1992.

[8] W. Kunz and D.K. Pradhan, "Multi-Level Logic Optimization by Implication Analysis," *Proc. ICCAD,* pp. 6-13, 1994.

[9] M. Schulz and E. Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," *Proc. FTCS*, pp. 30-34, 1988.

[10] E.Sentovich et. el. "SIS: A System for Sequential Circuit Synthesis" *Memorandum UCB M92/41, UC, Berkeley.*