Circuit Optimization by Rewiring

Shih-Chieh Chang (**Corresponding author**)
Department of Computer Science and Information Engineering
National Chung Cheng University.
Chiyai, Taiwan 621, Republic of China.
scchang@cs.ccu.edu.tw
011-886-5-2720411-6334 (Tel)
011-886-5-2720859 (FAX)

Lukas P.P.P. van Ginneken
Magma Design Automation Inc.
Mountain View CA 94043 Email
lukas@magma-da.com
Phone: (415)938-6970, x304 (Tel)

Malgorzata Marek-Sadowska
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
(805) 893 2721 (Tel)
(805) 893 3262 (Fax)
e-mail: mms@drum.ece.ucsb.edu

Circuit Optimization by Rewiring

Abstract

This paper presents a very efficient optimization method suitable for multi-level combinational circuits. The optimization is based on incremental restructuring of a circuit through a sequence of additions and removals of redundant wires. Our algorithm applies the techniques of Automatic Test Pattern Generation (ATPG) which can efficiently detect redundancies. During the ATPG process, certain nodes in the circuit must have particular logic assignments for a test to exist. Based on the properties of these mandatory assignments we have developed theorems to eliminate unnecessary wire redundancy checking. This results in significant performance improvement. The fast run time and the excellent scaling to large circuits make our Boolean optimization method practical for industrial applications.

Keywords: Rewiring, Logic Optimization, Logic Synthesis, Redundancy, and ATPG.

1 Introduction

The goal of logic synthesis is to realize, in some optimal way, a set of logic expressions using cells from a technology library. The process is typically divided into technology independent and technology dependent optimization. In this paper we discuss the problem of technology independent optimization for multi-level combinational Boolean networks. We assume that an initial multi-level Boolean network is given and it has to be optimized with respect to some cost function. The problem has been studied before and conventional methods are summarized in [11]. Because many of these methods do not scale well for large circuits, Automatic Test Pattern Generation (ATPG) based algorithms have become an attractive alternative [2] [5] [6] [8] [10] [13] [15] [16] [20]. In comparison to the majority of BDD based methods, an ATPG algorithm requires little memory to process large circuits. Although the running time of an ATPG algorithm may be exponential, the memory requirement is linear in the size of the circuit. The amount of computational effort spent in searching for test patterns can be accurately controlled. In addition, ATPG based optimizers can implicitly use circuit's observability and controllability don't cares without the need to calculate them explicitly.

ATPG based logic optimizers incrementally restructure a network without changing its functionality. The elementary operation is a single wire addition or removal. A wire which can be added or removed without changing the circuit's functionality is referred to as a redundant wire. One can consider redundant wires as a restructuring freedom during circuit optimization. Because ATPG allows us to detect redundant wires efficiently, most of the ATPG based algorithms optimize circuits by adding and removing redundant connections. For example consider the circuit [13] in Fig. 1. An ATPG algorithm can detect that a 2-input AND gate *m* and wire *e->m* will be redundant if added to the circuit. After adding *e->m*, two originally irredundant wires *e->g* and *a->f* will become redundant. Adding wire *e->m* and removing wires *e->g* and *a->f* results in a smaller circuit. This is the principle of the area optimization algorithms described in [6], [8] and [13]. Incremental restructuring achieved through wire addition and removal has many other applications. This technique can be used to remove particular wires to improve routability or performance [5]. In [9], circuit's structure is perturbed to improve partitioning.

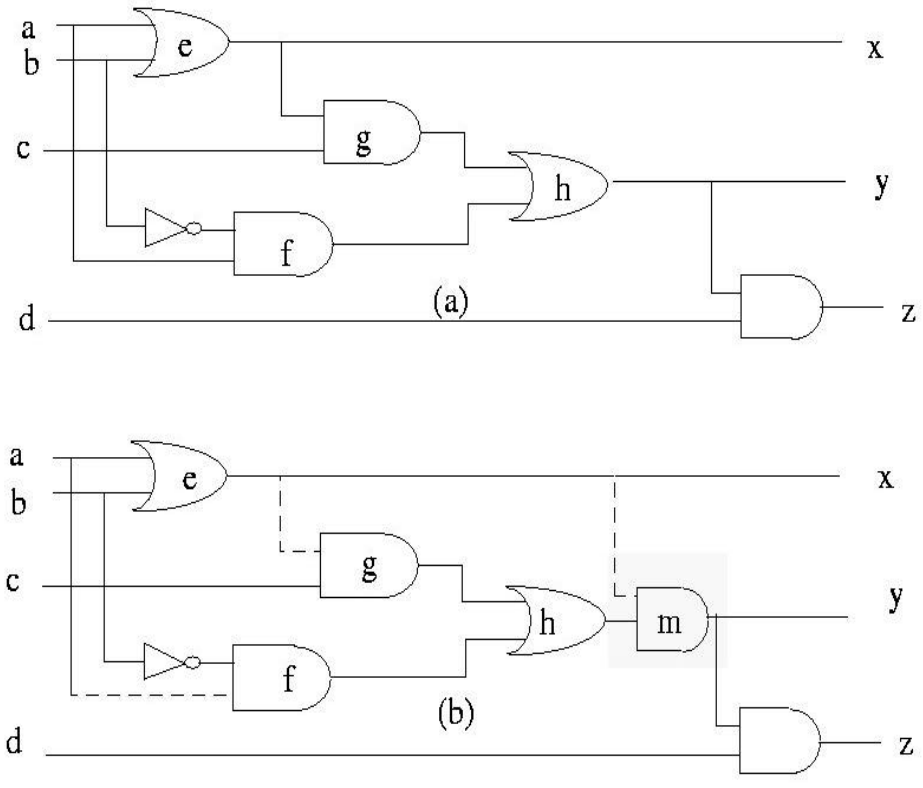


Fig. 1 Redundant wires caused by adding another redundant wire

In this paper we develop theory of single wire addition and removal. We investigate some necessary conditions for a wire to be redundant. These conditions are then used to improve and speed up the wire addition and removal process. The key question addressed here is which wires can be removed after adding a particular redundant wire.

To resolve this problem, we study characteristics of mandatory assignments [1], which must be satisfied for every test vector of a given fault. We introduce the *forced* and *observability* mandatory assignments. These two attributes of mandatory assignments can be computed efficiently with very little computational overhead. Based on the properties of mandatory assignments, we derive theorems that improve the results of ATPG based optimization. The speedup is achieved by eliminating many unnecessary redundancy checks on wires, which cannot be made redundant as a result of a particular incremental transformation.

This paper is organized as follows. In Section 2 we discuss previous work. Section 3 reviews ATPG concepts related to this paper. In Section 4 we introduce and discuss two important attributes of mandatory assignments, which are the *forced* mandatory assignment and the *observability* mandatory assignment. In Section 5 we derive some necessary conditions for a wire to become redundant. Section 6 describes the implementation of our algorithm. The efficiency is achieved by making use of the theorems described in Sections 4 and 5. Finally results and conclusions are presented.

2 Previous Work

In this section, we review several multi-level logic optimization algorithms [6] [8] [13] which all apply the principle of iterative addition of a redundant wire and/or gate followed by the removal of other redundant wires and/or gates. In all these works, ATPG techniques are used to determine whether a particular connection is redundant or not. Though the basic principle is about the same for most of the ATPG based logic optimization algorithms, they differ in where a redundant structure is added and what types of transformations are admissible. The speed of ATPG based multi-level logic optimizers depends on how efficiently the wires which become redundant after addition of a particular redundant wire, can be identified. The procedures to identify redundancies induced in the circuit by an

added redundant wire, are different in particular ATPG based optimizers. Note that when a redundant wire n_i ->n_j is added, a wire whose redundancy is caused by this change can be topologically "far" away from n_i ->n_j. After adding one wire, [8] tries to remove wires in the input cone of n_j, while [6] considers the input and the output cones. The algorithm [13], after adding one redundant wire to a circuit, applies redundancy test on the entire circuit. As stated in [17], though these ATPG algorithms are very powerful, their run times are unacceptable for large circuits because of many redundancy checks. In the following, we briefly summarize the strategies used in earlier algorithms.

In [8], the optimization algorithm first chooses a node *n* in a circuit. For each wire *w* in the input cone of node *n*, a search is performed to find new connections which when added to the circuit would cause *w* to be redundant. The wire *w* and the corresponding new connections will be stored in a table. Using this table, the algorithm selects and adds one new connection followed by the redundancy removal phase. This optimization process will iterate through all the nodes in the circuit.

The work [6] improves the results of [8] by allowing several new transformations such as adding two-input AND/OR/XOR gates to the circuit. Also, the search space is not confined to the input cone of any particular node. Techniques to filter out some of the unsuccessful transformations to reduce the computational effort are proposed.

In [13], each iteration step consists of adding one redundant wire or a redundant two-input AND/OR gate followed by the redundancy check in the entire circuit. Instead of building a table as in [8], a heuristic is used to decide where to add a redundant transformation. This heuristic, which is the key of the algorithm, is based on the observation that a circuit after optimization has less mandatory assignments of a certain type. This heuristic allows them to limit the choices of where to add a redundant transformation. Still, the time per iteration spent on whole-circuit redundancy removal can be very large.

3 Background and Definitions

In the following we review some standard logic synthesis terminology and ATPG related concepts which will be used throughout the paper. Here we consider only circuits

composed of AND, OR and INV gates. Complex gates can be handled by decomposing them into such gates.

If there is a directed path from a node n_i to a node n_j, we say that n_i is in *transitive fanin* of n_j and n_j is in *transitive fanout* of n_i. A *dominator* [12] *g* of a wire *w* is such a gate that all paths from *w* to any primary output have to pass through *g*. The value of an input to a gate is *controlling* if it determines the value of the gate's output regardless of the values on the other inputs. The controlling value is *1* for an OR gate, and *0* for an AND gate. The inverse of the controlling value is called the *noncontrolling* value or *sensitizing* value.

Consider a dominator of a wire $w=n_i->n_j$. The side inputs of a dominator are its inputs not in the transitive fanout of n_j. To generate a test for a stuck-at fault at wire *w*, all the side inputs of *w*'s dominators must be assigned their sensitizing values. In addition, to test a wire for a stuck-at-1 {0} fault, a test vector must generate the *activating* value of *0* {1} at the source node of the wire.

Let w_r be a wire tested for a stuck-at *0* {1} fault; the *faulty* circuit is the circuit in which w_r is replaced by a constant *0* {1}. An input combination *v* is a *test vector* if an output of the good (original) circuit and the faulty circuit are different when applying *v*. If no such a test vector exists, then the wire under stuck-at fault test is redundant.

The *mandatory assignments* (MAs) are the value assignments to nodes required for a test to exist and must be satisfied by any test vector. The process of computing these MAs and checking their consistency is referred to as *implication* and proceeds as follows. For a wire stuck-at fault, the MAs on the side inputs of the wire's dominators are set to the sensitizing values and the MA on the source node of the wire is set to the activating value. These MAs can then be propagated by using some simple rules such as if the output of an AND {OR} gate is *1* {*0*}, the inputs are *1* {*0*}. If all the inputs of an AND {OR} gate are *1* {*0*}, the output is *1* {*0*} etc. [1]. This process is called *direct implication*. More MAs can be found by more complicated approaches [14] [19]. During the process of direct implication, if the output of an AND {OR} gate is *1* {0}, all the inputs must be *1* {0}. We refer to this process as backward implication.

If the MAs of a stuck-at fault test cannot be consistent, the fault is untestable and therefore, the wire is redundant. A wire to be removed is referred to as the *target wire*. The corresponding stuck-at fault is called the *target fault*.

4 Forced Mandatory Assignments and Observability Mandatory Assignments

In this section, we discuss the *observability* and *forced* MAs. As mentioned in Section 3, an MA of a stuck-at fault can be implied from the MAs that activate the fault or sensitize a fault propagating path to one primary output. In the case of five-value logic [1], an MA can be *X*, *0*, *1*, *D* or $\overline{D}$.

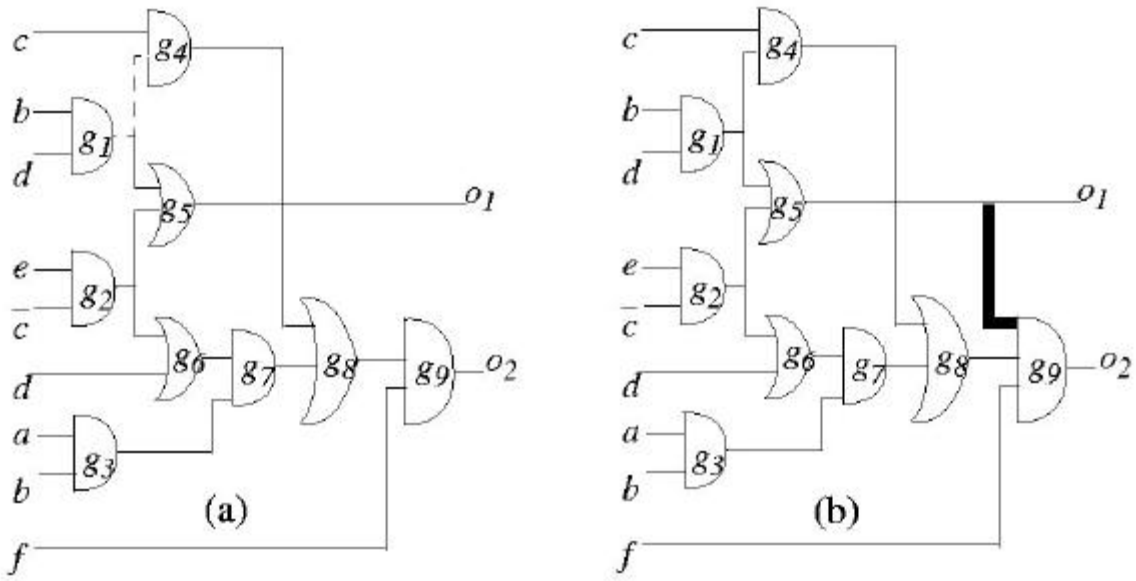


Fig. 2 Example of a single alternative wire.

We define an MA to be an *observability* MA if its purpose is to sensitize a fault-propagating path to one primary output during a stuck-at fault test. The observability MAs are those MAs that are necessary to make the fault observable at a primary output. The observability MAs are the MAs that are set to sensitize a fault propagating path but excluding the effect of the activating MA on the source of the fault. Therefore, the observability MAs are a subset of all MAs. Note that since the activating value does not play a role, the observability MAs do not depend on the source node of the target wire for which the test is sought. The set of observability MAs derived for the s-a-1 fault of a wire is the same as the set of observability MAs for the s-a-0 fault of the same wire; therefore the observability MAs can be defined for a wire (not for a particular fault).

For example, in the Fig 2(a), to make the g_1->g_4 s-a-1 fault observable at outputs, the MAs {$c=1$, $g_2=0$, $g_7=0$, $f=1$} must be set. These are the observability MAs. Note that MAs {$g_1=0$, $g_5=0$} are not observability MAs because they need to be derived from the activating MA $g_1=0$.

Let n be a node in a circuit C. Suppose that after a stuck-at fault test on wire w, node n has an MA. We build a new circuit $C'(n)$ as follows. (See Fig. 3.) If the MA of n is 1 or D, we disconnect n from its fanouts and connect a constant 0 to these fanouts. If the MA is 0 or $\overline{D}$, we connect a constant 1 to those fanouts. We say that node n has a forced MA in C if when we perform the same stuck-at fault test on wire w in $C'(n)$, the fault becomes untestable.

Conceptually, forced MAs cannot be changed if the fault is to remain testable. Non-forced MAs, on the other hand, are due to an incidental consequence of the test. For example in Fig. 2(a), consider the stuck-at-1 test for wire g_1->g_4. We have MAs {$g_1=0$, $c=1$, $g_2=0$, $g_5=0$, $g_7=0$, $f=1$, $g_4=D$, $g_8=D$, $g_9=D$}. The MA $c=1$ is a forced MA because disconnecting c from g_4 and inserting a 0 at an input of g_4 will make the stuck-at fault

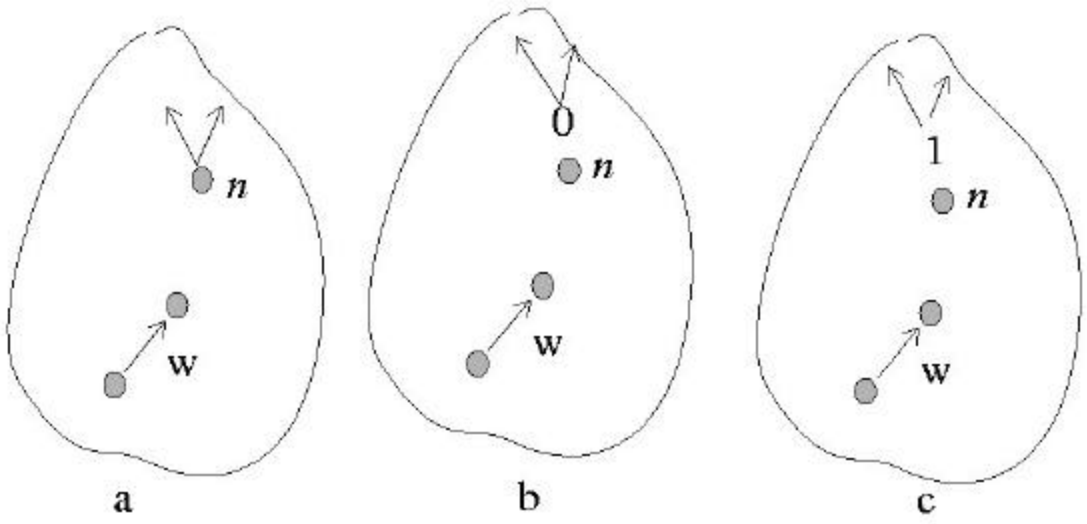


Fig. 3 Forced MAs. (a). w is the wire on which the stuck-at fault test is performed. (b) shows $C'(n)$ when n has a MA=1 or D. (c) shows $C'(n)$ when the node has a MA=0 or $\overline{D}$.

untestable. The MA $g_5=0$ is non-forced because after disconnecting g_5 from o_1 and connecting 1 to o_1, the fault g_1->g_4 s-a-1 is still testable.

For a wire stuck-at fault test, forced MAs can be computed directly from the definition. However such an approach is too time-consuming. In the following we discuss how to efficiently determine forced MAs.

Lemma 1: The MAs obtained by setting the side inputs of the dominators to their non-controlling values, setting the outputs of the dominators to D or $\overline{D}$, and setting the *activating* MA on the source node of the target wire stuck-at fault, are all forced. In addition, the MAs that are set from backward implication of other MAs are also forced.

Proof. A controlling value in a side input of a dominator will cause the dominator to have a constant value. This will block the fault propagation and result in untestability of the target fault. Therefore, the uncontrolling value in a side input of a dominator is a forced MA. The proof is similar for the activating MA and the MAs derived from backward implication. **QED**.

Lemma 1 suggests that whether an MA is forced or not can be determined while performing direct implication. Therefore, no additional test is required to decide whether an MA of a target fault is forced or not. For example, in Fig. 2(a) consider again g_1->g_4 stuck-at-1 fault. We have MAs {$c=1$, $g_7=0$, $f=1$} as forced MAs because they are side inputs of dominators, and {$g_2=0$, $g_5=0$} are not forced MAs because they are obtained from forward propagation of other MAs.

Now, we discuss how to obtain forced MAs in a learning process which applies direct implication recursively. During recursive learning, a node has an MA if in all decisions, the node has the same value assigned. Similarly, during recursive learning, a node has a forced MA if in all decisions, the node has the same forced MA.

Consider the example in Fig. 1a. When the MA of h is *1*, either the MA of g is *1* or the MA of f is *1*. When the MA of g is *1*, we have MA $c=1$ and MA $e=1$. When the MA of f is *1*, we have MA $b=0$, and MA $a=1$. Propagating MAs $\{b=0, a=1\}$ forward yields MA $e=1$. Since the MA of e is *1* in both decisions, the MA of e is *1*. However, this MA is not forced because in one of the cases, MA $e=1$ is obtained as a result of forward propagation.

If a forced MA is changed, the fault becomes untestable. For example in Fig. 2(b), node g_9 (a dominator) has a forced MA D in g_1->g_4 s-a-1 test. In the circuit, suppose we know that wire g_5->g_9 is redundant. If the redundant wire g_5->g_9 is added to the circuit, the forced

MA of g_9 is changed from *D* to *0* because g_5 has the MA of *0* in g_1->g_4 stuck-at-1 test. As a result, g_1->g_4 is redundant after adding g_5->g_9.

5 Wires Which Can Never Be Redundant After Adding One Redundant Wire

In this section, we consider the problem of determining which wires can never become redundant after adding **one** redundant wire. Before the optimization process by adding and removing redundancy, the circuit may already contain some redundant wires. After adding one redundant wire to the circuit, some wires may become redundant. Here, we distinguish the wires that were already redundant before applying a redundant transformation, from the wires which become redundant as a result of this transformation. For example, in Fig. 1b, wires *e->g* and *a->f* are the newly redundant wires after adding *e->m*. All the theorems below refer to the redundant wires caused by the addition of a single redundant wire and redundant AND/OR gate as shown in Fig. 1b.

To simplify the discussion, we make the following assumptions. We assume that the circuit before applying the redundant transformation does not contain redundant wires. The reason is that if there were redundant wires which were not the consequence of adding a redundant wire, we would have to distinguish between *previously redundant wires* and *newly redundant wires*. Our theorems in the following are intended to identify the wires that become redundant due to addition of one redundant wire, i.e. newly redundant wires. In order not to confuse them with the previously redundant wires, we assume there are no redundant wires in the circuit at the beginning of the process. On the other hand, if there were previously redundant wires present in an initial circuit, our procedures would not detect them because based on our theorems they would not be declared as possible newly redundant wires. Also, all the theorems are applied as a form of filters to speedup the process. When a wire is claimed to be possibly redundant, the wire still requires redundancy check before its removal. Therefore, the presence of redundant wires in an initial circuit will not affect the correctness of our final results.

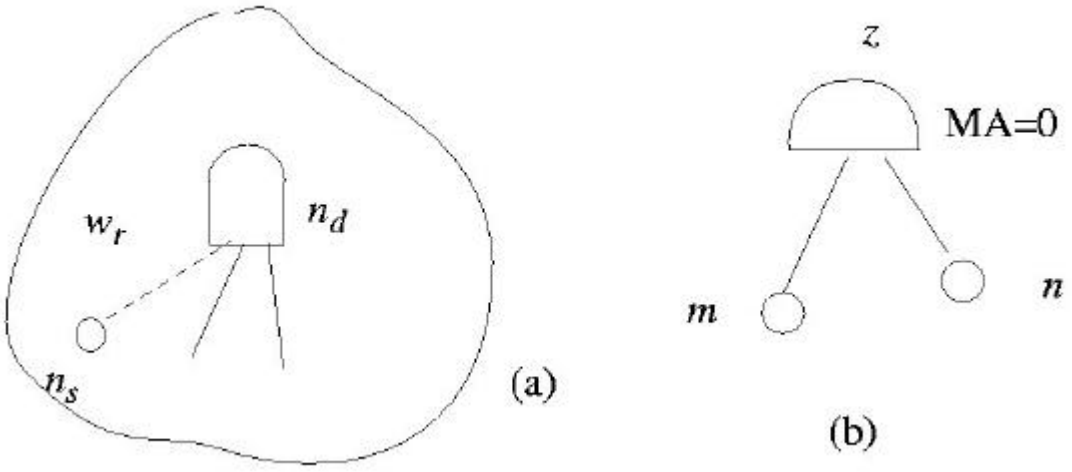


Fig. 4 Adding a redundant wire w_r in an irredundant circuit.

Additionally we do not consider the problem of *two simultaneously redundant wires* [6]. Two wires are simultaneously redundant if we can add one and remove the other at the same time but we cannot add one without changing the circuit's functionality and then remove the other.

Let $(C \cup w_r)$ denotes a circuit C with an added wire w_r and let $(C \backslash w_t)$ be a circuit obtained by removing wire w_t from the circuit C. Without loss of generality, consider an irredundant circuit C and a redundant wire $w_r = n_s \text{->} n_d$ to an AND gate is added in C as depicted in Fig. 4(a). Since w_r is redundant, computation of MAs for the w_r s-a-1 test leads to inconsistency. In all the following lemmas, w_r denotes a redundant wire which has been added to an irredundant circuit C and w_t is an irredundant wire in C which becomes redundant in the new circuit $C \cup w_r$. Note that the MAs for a redundant wire are inconsistent and have no meaning. However, the observability MAs for a redundant wire may not be inconsistent. The following lemma shows that with our assumption, the observability MAs for the newly added redundant wire must be consistent.

Lemma 2: Let $w_r = n_s \text{->} n_d$ be a redundant wire if added to C. The *observability* MAs for the redundant wire w_r must be consistent in C.

Proof. Let n_d be an AND {OR} gate. Suppose the observability MAs are inconsistent. Any new connection which does not exist in C and is a fanin of n_d is redundant. Therefore, a

constant *0* {*1*} that fanins to n_d is also redundant. We can then conclude that n_d can be replaced by a constant *0*{*1*} which contradicts the assumption of the initial irredundancy of *C*. **QED**.

The procedure of computing observability MAs can be done by first assigning the non-controlling MAs on the side inputs of dominators and then propagating those MAs to imply more observability MAs. Lemma 2 states that during propagation, the observability MAs for a redundant wire will never be inconsistent. Therefore there will be no conflict of assigning observability MA to a node. This lemma will be applied in the proof of the next lemma.

Lemma 3: Let n_d be an AND {OR} gate. The wire w_r=n_s->n_d, if added, is redundant, if and only if n_s has an observability MA *1 {0}* for w_r stuck-at fault test. (A similar theorem is shown in [13].)

Proof. Let's first compute the observability MAs for wire w_r=n_s->n_d. Based on Lemma 2, these MAs are consistent. Then, assign *0 {1}*, the activating value at n_s. Because wire w_r is redundant, this MA *0 {1}* assigned at n_s must be inconsistent with the previous assignment. Therefore, when computing observability MAs, node n_s must have had an observability MA of *1 {0}*. **QED.**

Using Lemma 3, one can find all the redundant wires which connect to a particular node in a circuit. For example, in Fig. 1, the observability MAs for (*any node)->m* stuck-at-1 test are {*e=1, h=1*}. The MA of node *e* is *1* and node *m* is an AND gate. Therefore, according to Lemma 3, wire *e->m* if added, is a redundant wire because activation of wire *e->m* s-a-1 fault requires *e* to be set to *0*. For another example, in Fig. 2, one can find that g_5 has an observability MA of *1* for wire *(any node)->*g_9. Therefore, wire g_5->g_9 is redundant.

Lemma 4: Suppose an irredundant wire w_t in *C* becomes redundant after adding a redundant wire w_r = n_s->n_d. Then wire w_r is irredundant in ($C \cup w_r \backslash w_t$).

Proof. If w_r is still redundant after removing w_t, wire w_t is redundant in *C* without adding w_r. This conflicts with our original assumption that *C* is an irredundant circuit. **QED**.

Let $w_r = n_s\text{->}n_d$ be a redundant wire added to C, and w_t be a wire which becomes redundant after this change. According to Lemma 3, node n_s must have an observability MA, for $w_r = n_s\text{->}n_d$ stuck-at fault test in C and let its value be *ma*.

Lemma 5: In the new circuit $C \cup w_r \backslash w_t$, the observability MA at n_s for w_r must have a different value from the previous value *ma*. In other words, if the observability MA at n_s for w_r in $C \cup w_r$ is the same as the MA at n_s when computed in $C \cup w_r \backslash w_t$, wire w_t cannot be redundant in $C \cup w_r$.

Proof. If node n_s has the same observability MA, w_r is still redundant after removing w_t. This contradicts Lemma 4. **QED**.

For example, in Fig. 1b, observability MAs for *e->m* are {*e=1, h=1*}. Suppose we remove *e->g*. Observability MAs for *e->m* become {*h=1*}. The observability MA at *e* is not *1* as before; i.e. it is different. Therefore, after removing *e->g*, wire *e->m* is no longer redundant.

By computing the observability MAs of wire (*any node*)$\text{->}n_d$, we can find which wires connecting to n_d are redundant and can be added to a circuit (from Lemma 3). If the complete set of observability MAs is found, we can obtain the complete set of such new redundant wires connecting to n_d. However, finding all observability MAs can be very time consuming. A heuristic implication algorithm, which may only traverse some wires/nodes, may find only a subset of all observability MAs and as a result only a subset of new redundant wires for possible addition will be detected. Although the redundancy of a wire is defined independently of an implication algorithm, some redundancies can be easy to detect, while detection of other redundancies may require more computational effort. Suppose that while applying implication to detect that a wire w_r will be redundant if added, a set of wires has not been traversed. In the next theorem we will show that none of these wires can be redundant when w_r is added to the circuit.

Theorem 6: Suppose an implication algorithm is invoked and based on Lemma 3, we find that a wire w_r if added will be redundant. If a wire w is not visited during the process of implication which detects redundancy of w_r, then after adding w_r, the wire w will not become redundant.

Proof. Removing the wire w cannot change the observability MA at n_s. **QED**.

Example 1: In Fig. 1(b), suppose we would like to add a redundant wire to node *m* (*m* is the highlighted AND gate) and assume wire *e->m* is not present in the figure. When computing the observability MAs for *(any node)->m*, we first set *h=1* to propagate the fault *D* from *m* to output *y*. Since the MA of *h* is *1*, either the MA of *g* is *1* or the MA of *f* is *1*. If the MA of *g* is *1*, we have MAs {*c=1, e=1*}. If the MA of *f* is 1, we have MAs {*a=1, b=0, e=1*}. Therefore we conclude *e=1*. Because that the MA of *e* is *1* and node *m* is an AND gate, according to Lemma 3, we find that wire *e->m* is redundant and can be added to the circuit. Now, we would like to use Theorem 6 to detect wires which cannot become redundant after adding *e->m*. In this example, we assume MA *g=1* or *f=1* from *h=1*. For *g=1*, the wires {*g->h, c->g, e->g*} are traversed and for *f=1*, the wires {*f->h, a->f, a->e, b->f, b->e*} are traversed. Since the wires {*d->z, m->z*} are not visited, they cannot become redundant according to Theorem 6.

Theorem 7: Let the wire w_r, if added, be a redundant wire. After obtaining the observability MAs for w_r, let us consider a wire w_a =*m->z*. If node *z* is an AND (OR) gate with an **observability forced** MA *0 (1)*, and node *m* has no MA, then, we can conclude that wire w_a cannot become redundant in ($C \cup w_r$).

Proof. According to Lemma 5, if the observability MA at n_s for wire w_r=n_s->n_d in $C \cup w_r \backslash w_a$ is the same as in $C \cup w_r$, wire w_a=*m->z* is not redundant in ($C \cup w_r$). Without loss of generality, let us consider a two-input AND gate *z* which has an observability forced MA *z=0* for wire w_r=n_s->n_d and one of its input *m* has no MA. See Figure 4b. There are two possible combinations of MAs at the inputs of *z*: (*m, n*) = (*X, 0*), or (*X, X*). We discuss them separately and prove that removing wire *m->z* will not change the observability MA at n_s.

Case 1: (*m, n*) = (*X, 0*). Removing wire *m->z* will not change any MA in the circuit so wire *m->z* is not redundant after adding w_r.

Case 2: (*m, n*) = (*X, X*). Removing wire *m->z* will cause the MA of node *n* to be *0*. Though more MAs can be implied by MA *n=0*, it will not change other observability MAs in the circuit including the MA at n_s. Therefore, the wire *m->z* is not redundant in ($C \cup w_r$). **QED**.

Consider Example 1 again in Fig. 1. After computing the observability MAs for wire (*any node)->m*, node *h* has a forced observability MA of *1*. In addition, node *g* and node *f*

do not have any MA. According Theorem 7, if we add any redundant wire to node *m*, wires *g->h* and *f->h* will not become redundant. Therefore, if a redundant wire is added to node *m*, the only possible redundant wires are {*a->f, b->f, c->g, e->g, b->e, a->e*}

Theorem 8: The wire $w=n_x->n_z$ is not redundant in $(C \cup w_r)$ if n_z is an AND {OR} gate which does not have an MA of value *1* {0} and n_x has an MA of *1* {0}.

Proof. Removing *w* will not change the observability MAs so wire *w* is not redundant in $(C \cup w_r)$. **QED**.

Again in Example 1, let us look at the circuit in Fig. 1. To add a redundant wire to node *m*, we have the observability MA *h=1* from which, we can deduce either *f=1* or *g=1*. When considering *f=1*, we have MAs {*a=1, b=0*}. According to Theorem 8, we know that wire *b->e* cannot be redundant for whatever redundant wire added to node *m*. If a redundant wire is added to node *m*, the possible redundant wires are {*a->f, b->f, c->g, e->g, a->e*}.

Theorem 9: Suppose node n_z is an AND {OR} gate and $n_x->n_z$ is one of n_z's input wire. If n_x has an observability MA of *0* {*1*} for w_r, then all the other input wires of n_z are not redundant in $(C \cup w_r)$.

Proof. The proof is similar to Theorem 8. **QED**.

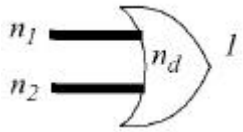
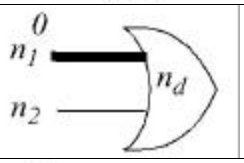
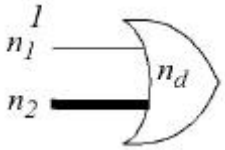
Theorem 10: If removing a wire *w* does not change the observability MAs for w_r which were found applying recursive learning then wire *w* cannot be redundant after adding w_r.

Proof. If the observability MA does not change, according Lemma 2, the wire will not become redundant. **QED**.

In Example 1, MA *e=1* is an observability MA for *e->m*. Since removal of *b->f* or *c->g* does not change the MA, therefore based on Theorem 10, the wires {*b->f* , *c->g*}cannot be redundant.

TABLE

Theorem	Condition	Descriptions
6		A wire is not redundant if it is not visited during an implication process.

7		The bold wires $n1$->nd and n_2->n_d are not redundant if nd has a forced observability MA of *1*.
8		The bold wire $n1$->nd is not redundant if n_1 has observability MA of *0* and n_d has no MA.
9		The bold wire n_3->n_d is not redundant if n_1 has observability MA of *1*.
10	Do not change the observability MAs	A wire *w* is not redundant if removing *w* will not change the observability MAs.

Let us consider again to add the redundant wire *e->m* in the circuit in Fig. 1.b. We summarize our procedure of pruning irredundant wires in the following.

1. Based on Theorem 6, the wires {*d->z, m->z*} cannot be redundant.

2. From Theorem 7, the wires {*g->h, f->h*} cannot be redundant.

3. From Theorem 8, the wire {*b->e*} cannot be redundant.

4. Based on Theorem 10, the wires {*b->f, c->g*} cannot be redundant. As a result, if the redundant wire *e->m* is added, the possible redundant wires are {*a->f, e->g, a->e*}.

All the above theorems specify some conditions to identify wires, which cannot become redundant after adding a redundant wire w_r**.** Our optimization routine can use these theorems as a filter to avoid unnecessary redundancy checks. All theorems are summarized in TABLE 1 for the case of OR gates. A similar table can be built for the case of AND gates.

6 Implementation

The pseudo code of our implementation is shown in Fig. 5. This algorithm extends the ideas of [6]. First the algorithm visits each node in the circuit. In each iteration, we arbitrarily choose a node n_d where a new redundant wire w_r (n_s->n_d) will be connected. The optimization routines will try to find a "good" w_r such that its addition can cause many

wires to be redundant. In order to find a "good" w_r, we first determine a set of wires, which can possibly be removed if w_r is connected to n_d. Initially this set contains all the wires whose transitive fanout has an intersection with the fanout cone of n_d. On these wires, we apply Theorems 6-9 to prune the wires which cannot become redundant. After pruning, on the remaining wires in the set, we attempt to find their alternative wires [6] and build a table listing the alternative wire relationship as mentioned in Section 2. Finally, we choose and add one redundant wire to the circuit and remove as many as possible of its alternatives listed in the table.

```
foreach node n_d in the circuit {
  Find the observability MAs of wire (any node->n_d);

  Construct an array of wires, array_wires, which contain wires in the fanin and fanout cone
   and some wires which are within k levels in the transitive fanin of a dominator of n_d.

  for each wire w_t in the array_wires {
    Use Theorems 6-10 to skip wire w_t which cannot become redundant;
    Calculate the MAs of w_t
    Use Theorem 12 of [7] to check if a wire can be added to n_d to make w_t redundant
    Fill source_array with nodes which have an MA using Theorem 11 of [7];
    /* the source array contains nodes, which will become the source node of an added wire */
    Use Theorem 14 of [7] to prune source_array
   }
   optimize the circuit as in [6]
 }
```

Fig. 5 The algorithm.

7 Experimental Results

TABLE 2 compares the optimization results of SIS1.1 [4] [18], HANNIBAL[13] and ours (termed REWIRE) for some benchmark circuits. Since our circuits are in the form of AND and OR gates, we post-process those circuits with "el; sweep; el; simplify" (which are commands in SIS) and compare the factored literal count with SIS and HANNIBAL. Column 2 and 6 show the run time and factor literal count for script.boolean and Column 3 and 7 show the run time and results for script.rugged. The run time and results for HANNIBAL are shown in column 4 and 8. Our run time and results are shown in column 5 and 9. From the circuits shown in the TABLE 2, REWIRE is 126 times faster than

HANNIBAL. The literal counts are about the same for REWIRE and HANNIBAL. Our results are also 17% better than script.boolean and 14% better than script.rugged from SIS1.1. The first section of the TABLE 2 lists and compares the circuits reported in [13]. The remainder of the table lists some additional circuits. We also like to mention that the BDD based optimization may have large run time even for small circuits. For example, *script.rugged* requires 62476 seconds to run circuit *too_large*, which has 300 literals while REWIRE only takes 31.9 seconds. All experiments were done using the SUN workstation, Solbourne Series 6/006 50M Hz.

8 Conclusions

In this paper we have considered the problem of determining which wires become redundant after inserting one redundant wire to an originally irredundant circuit. We have developed theory to answer this question. Based on the properties of redundant wires we have proposed and implemented an efficient Boolean optimizer called Rewire. As demonstrated in our experiments, Rewire is much faster then the optimization algorithm [13], with competitive results. Further speedup is possible because our implementation did not use Theorem 6. Optimization results on some very large circuits, such as s38417, suggest that our approach scales well. The above theorems are applicable to other ATPG based optimizations also. It seems likely that the algorithms of [2] [5] [8] [13] [16] [20] can take advantage of our theorems to improve their run times.

9 Acknowledgment

The first author acknowledges support from the National Science Council in Taiwan through the Grant NSC 86-2213-E-194-025-T. The third author acknowledges support from the National Science Foundation through the Grant MIP 9419119.

10 References

[1] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.

[2] C. L. Berman and L. H. Trevillyan. “Global Flow Optimization in Automatic Logic Design,” IEEE Trans. CAD 10, pp. 557-564, May 1991.

[3] M. R. C. M. Berkelaar, L. P. P. P. van Ginneken: "Efficient Orthonormality Testing for Synthesis with Pass-Transistor Selectors", *Digest Int. Conf. on Computer Aided Design*, pp. 256-263, Nov. 1995.

[4] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: Multi-level Interactive Logic Optimization system," IEEE Trans. on CAD Vol. 6, pp. 1062-1081, Nov. 1989.

[5] S.C. Chang, K.T. Cheng, N.S. Woo, M. Marek-Sadowska, " Post Layout Logic Restructuring Using Alternative Wires ", IEEE Trans. on Computer Aided Design, Vol. 16, pp.587-596, June 1997.

[6] S.C. Chang, M. Marek-Sadowska, and K.T. Cheng, "Perturb and Simplify: Multi-level Boolean Network Optimizer", IEEE Trans. on Computer Aided Design, Vol. 15, pp. 1494-1504, Nov. 1996.

[7] S.C. Chang, L. P.P.P. van Ginneken, and M. Marek-Sadowska, "Fast Boolean Optimization by Rewiring," *Proc. Int. Conf. on Computer Aided Design*, pp.262-269, Nov. 1996.

[8] K. T. Cheng and L. A. Entrena, "Multi-Level Logic Optimization by Redundancy Addition and Removal," in *Proc. European Conference On Design Automation*, pp. 373-377, Feb. 1993.

[9] D. I. Cheng, C. C. Lin and M. Marek-Sadowska, "Circuit Partitioning with Logic Perturbation," in *Proc. Int. Conference on Computer Aided Design*, pp., 650-655, Nov. 1995.

[10] L. A. Entrena and K. T. Cheng, "Sequential Logic Optimization By Redundancy Addition and Removal", *Proc. Int. Conf. on Computer Aided Design*, Nov. 1993.

[11] G.Hachtel and F.Somenzi, "Logic Synthesis and Verification Algorithms", Kluwer Academic Publishers, 1996.

[12] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm For ATPG," *Proc. Design Automation Conf.*, pp. 502-508, June 1987.

[13] W. Kunz and D.K. Pradhan, "Multi-Level Logic Optimization by Implication Analysis", *Digest Int. Conf. on Computer Aided Design*, pp. 6-13, Nov.1994.

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

[15] B. Rohfleisch, F. Brglez, "Introduction of Permissible Bridges with Application to Logic Optimization after Technology Mapping," Proc. Edac, pp. 87-93, 1994.

[16] B. Rohfleisch, B. Wurth, K. Antreich "Logic Clause Analysis for Delay Optimization", Proc. DAC, pp. 668-672, 1995

[17] R. Rudell, "Tutorial: Design of a Logic Synthesis System", Proc. DAC, pp. 191-196, 1996

[18] H. Savoj, H.Y. Wang, and R.K. Brayton, "Improved Scripts in MIS-II for Logic Minimization of Combinational Circuits," Proc. IWLS, 1990.

[19] M. Schulz and E. Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," *Proc. Fault Tolerant Computing Symp.*, pp. 30-34, June 1988.

[20] M. Yuguchi, Y. Nakamura, K. Wakabayashi, T. Fujita "Multi-Level Minimization based on Multi-Signal Implications", proc. DAC, 1995, pp. 658-662.

TABLE 2

Circuits	**SIS-1.1 Boolean (sec)**	**SIS-1.1 rugged (sec)**	**Hannibal cpu (sec)**	**Rewire cpu (sec)**	**SIS-1.1 Boolean (lits)**	**SIS-1.1 rugged (lits)**	**Hannibal (lits)**	**Rewire (lits)**
C3540	133.5	171.2	6815	85.6	1299	*1288	1154	1127
C432	6.2	700.7	95	2.0	240	205	161	171
C2670	38.8	183.4	1782	31.7	759	*746	718	697
C880	11.6	36.3	269	3.2	427	415	417	415
C5315	102.1	158.1	15611	65.9	1815	1734	1760	1687
C1355	16.4	123.1	555	17.5	554	552	544	552
C6288	425.1	378.3	13704	89.0	3550	*3337	3240	3251
C1908	22.2	119.3	935	16.0	552	540	517	512
C499	11.0	107.6	543	8.4	554	552	544	550
subtotal	**767**	**1978**	**40309**	**319.3**	**9750**	**9395**	**8113**	**8107**
relative	**2.4**	**6.2**	**126.24**	**1**	**1.17**	**1.14**	**1.001**	**1**
s13207	FA	FA		541.7	FA	FA		2719
s38417	FA	FA		1746.6	FA	FA		10434
s5378	159.0	205.4		77.8	1471	*1438		1351
s9234	200.2	264.0		126.1	1943	*1943		1724
alu2	77.5	160.4		50.3	446	361		324
alu4	282.0	596.3		152.1	880	698		623
term1	16.0	25.0		5.0	237	168		145
too_large	1081.9	62476.1		31.9	437	302		301
ttt2	6.6	9.1		5.2	223	215		179
z4ml	1.0	1.3		0.5	48	45		36
f51m	2.9	4.4		4.8	135	91		105
frg2	63.6	115.0		85.1	933	893		761

FA: Fatal *: Cannot complete due to BDD nodes out of space.