Gate Mapping Automation for Asynchronous NULL Convention Logic Circuits

Farhad A. Parsan, *Student Member, IEEE*, Waleed K. Al-Assadi, *Senior Member, IEEE*, and Scott C. Smith, *Senior Member, IEEE*

***Abstract*— Design automation techniques are a key challenge in the widespread application of timing-robust asynchronous circuit styles. In this paper, a new methodology for mapping multi-rail logic expressions to a NULL convention logic (NCL) gate library is proposed. The new methodology is then compared to another recently proposed mapping approach, demonstrating that the new methodology can further reduce the area and improve the delay of NCL circuits. Also, in contrast to the original approach, which only targets area reduction, the new methodology can target any arbitrary cost function or use any subset of the NCL gate library for mapping. In order to automate the new methodology and compare it with the original one, both methodologies were implemented in the Perl programming language and compared in terms of mapping performance and runtime. The results show that, depending on the test circuit, the new methodology can offer up to 10% improvement in area, and 39% improvement in delay.**

***Index Terms*— Automation, factoring, gate mapping, grouping, NULL convention logic (NCL), technology mapping.**

I. Introduction

ASYNCHRONOUS logic design paradigms such as NULL convention logic (NCL) [1]–[7] offer many advantages over the conventional synchronous designs. These advantages include inherent robustness against power supply, temperature, and process variations; less noise/electromagnetic interference; and easy design reuse. NCL circuits are correct-by-construction designs [2], requiring very little timing analysis, if any. In today's nanometer processes where meeting timing closure is becoming more and more difficult due to increasing clock rates and decreasing feature sizes, this quality can be very attractive. NCL is also one of the few asynchronous design paradigms that have been used for a number of industrial designs [8], [9].

The main obstacle preventing the widespread application of asynchronous design paradigms, such as NCL, in industry is the lack of standard computer-aided design (CAD) tools that support automating the design process. Several NCL design automation flows have been proposed in the literature so far [4]–[7]. Technology mapping, in particular, is the heart of such design flows since the performance of the mapped circuits is directly determined by the efficiency of such mapping. In this paper, we address the problem of mapping multi-rail logic expressions to an NCL gate library, which is a form of technology mapping for a certain class of NCL design flows. Mapping multi-rail logic expressions to NCL gates is also an essential part of a custom NCL design flow, and, in contrast to mapping a Boolean logic expression to a limited set of basic Boolean gates, an efficient manual mapping of multi-rail logic expressions to 27 NCL gates is not trivial, especially when the logic expressions contain many product terms (explained later). In this paper, we propose a multi-rail logic expression mapping algorithm and compare it with the only other such mapping algorithm in the literature. We show that our proposed algorithm can outperform the existing one in several ways.

Manuscript received June 26, 2012; revised November 9, 2012; accepted December 8, 2012. Date of publication March 12, 2013; date of current version December 20, 2013. This work was supported in part by the NSF under Grant CCLI.

F. A. Parsan and S. C. Smith are with the Electrical Engineering Department, University of Arkansas, Fayetteville, AR 72701 USA (e-mail: fparsan@uark.edu; smithsco@uark.edu).

W. K. Al-Assadi is with the Electrical and Computer Engineering Department, University of South Alabama, Mobile, AL 36688 USA (e-mail: waleed@usouthal.edu).

Digital Object Identifier 10.1109/TVLSI.2012.2231889

A. NCL Overview

NCL is a self-timed logic paradigm in which control is inherent in each datum. NCL uses delay-insensitive codes [10] for data communication, alternating between set and reset phases. In the set phase, data changes from spacer (called NULL) to a proper codeword (called DATA); and in the reset phase it changes back to NULL. NCL combines DATA and NULL into a mixed path, usually represented by dual-rail or quad-rail signals. A dual-rail signal D consists of two wires, D^0 and D^1. D is logic 1 (DATA1) when $D^1 = 1$ and $D^0 = 0$; it is logic 0 (DATA0) when $D^0 = 1$ and $D^1 = 0$, and is NULL when both D^0 and D^1 are 0. NCL uses state-holding threshold gates with hysteresis [11]. An NCL gate is generally denoted as THmnW$w_1, \ldots, w_n$ where n is the number of inputs, m is the threshold of the gate, and $w_1, w_2, \ldots, w_n$ are the weights of inputs when they are greater than1. Assuming that the inputs are $x_1, \ldots, x_n$, then the output of an NCL gate is asserted when $x_1 w_1 + \cdots + x_n w_n d \geq m$ (see Table II for the set function of all the NCL gates). Since NCL gates have hysteresis, once the set function of a gate is satisfied and its output is asserted, it remains asserted until all its inputs are deasserted.

NCL circuits must be input-complete and observable in order to preserve delay-insensitivity [3]. The input-completeness criterion states that outputs of a combinational circuit may not transition to DATA (NULL) before all the inputs have transitioned to DATA (NULL). However, according to the weak conditions of Seitz's delay-insensitive signaling [12], in circuits with multiple outputs, it is acceptable for some outputs to transition to DATA (NULL) without having a complete input DATA (NULL) set as long as all outputs

cannot transition to DATA (NULL) before all inputs transition to DATA (NULL). Observability, on the other hand, requires that no orphans may propagate through a gate [13]. An orphan is a signal transition on either a wire (wire orphan) or the output of a gate (gate orphan) that is not acknowledged by any primary output of a circuit. An output is said to acknowledge a signal transition if it always transitions following that signal transition. Wire orphans are not considered serious and can be ignored by assuming isochronic fork conditions [14], [15], but gate orphans can cause delay sensitivity problems, and therefore must be avoided during circuit design.

B. Existing NCL Design Automation Flows

Several NCL design automation flows have been proposed in the literature [4]–[7]. Any NCL design flow must address two main issues: synthesizing a synchronous register-transfer level (RTL) design into an NCL netlist without violating input-completeness and observability; and optimizing the synthesized circuit under a predefined cost function. All existing NCL design automation flows can convert a synchronous RTL design into an input-complete and observable NCL netlist; however, they differ in their technology-mapping approach and optimization level.

The design flow in [4] starts with a synchronous RTL design written in VHDL. The RTL design must explicitly specify the location of registers, either through inference or instantiation. These registers are later replaced with NCL registers, and the required handshaking signals and completion detection components are added. The rest of the RTL design that describes the combinational blocks can be a synthesizable behavioral description. The RTL design can be functionally verified by simulating it in any VHDL simulator using a single-rail multivalued signal type (NCL_LOGIC) that includes NULL ("N" value). After verification, the combinational logic is synthesized into a so-called "3NCL" gate-level netlist comprised of only two-input Boolean gates using Synopsys design compiler. At the time of synthesis, design compiler treats NULL ("N" value) as a *don't-care* so it does not affect the synthesized circuit. Next, the single-rail multivalued signals in the 3NCL netlist are converted to dual-rail signals, and each two-input Boolean gate is macro-expanded to its DIMS style [16] equivalent. The resultant circuit is now called a "2NCL" circuit comprised of only two-input C-elements (i.e., TH22) [17] and OR gates. A 2NCL circuit built this way is proved to be input-complete and observable by design [18], [19], but it is not optimized. The 2NCL circuit is then optimized by replacing each two-input C-element with its Boolean set phase equivalent (i.e., a two-input AND gate); and a multilevel logic minimization is performed by using the design compiler again. The resulting so-called smoothed and optimized netlist is finally mapped to NCL complex gates by running design compiler for the last time. This NCL design flow heavily depends on the sophisticated synchronous synthesis tools (design compiler, here) and only targets area reduction for optimization.

In [5], systematic and general technology mapping and cell merger methods are introduced that can be used for designing all types of timing-robust asynchronous threshold networks, including NCL circuits. These methods are able to optimize both the datapath and control portions of NCL circuits, and they can target either area or delay or a combination of the two as the cost function. However, these methods are not a comprehensive NCL design flow (i.e., starting from an RTL description to an NCL netlist) but are postprocessing optimization steps that can add to other NCL design flows, such as the one in [4], to further optimize area or delay of the final circuit. In other words, the input to these methods is an already synthesized and working NCL circuit that has not been optimized yet.

The NCL design automation flow in [6] is very different from other existing NCL design flows in that it relies much less on the synchronous synthesis tools. This design flow is loosely based on the TCR method explained in [9]. Similar to the design flow in [4], it starts with a synchronous RTL design in which registers are explicitly defined; in contrast, the designer also has to partition the combinational blocks into combinational modules with appropriate sizes. These combinational modules can be described behaviorally but must be connected in a structural manner to form the entire combinational block. The combinational modules are initially synthesized into Boolean sum-of-products (SOP) expressions, such that each output of a combinational module is described in terms of its inputs. This step can be performed with just about any synthesis tool. The resultant Boolean SOP expressions are then converted to dual-rail SOP expressions and minimized using custom scripts. Finally, a minimum number of the minimized dual-rail SOP expressions are selected to become input-complete (according to the weak conditions of Seitz [12]) and the final minimized and input-complete SOP expressions are mapped to NCL gates by using a custom mapping algorithm, to be discussed in detail later.

Finally, the Unified NCL Environment (UNCLE) [7] is a very recent interesting work on automating the NCL design flow. Conceptually, this flow is similar to the one in [4] but with more flexibility. The design flow starts with Verilog RTL code, which is initially synthesized to a library of Boolean gates, such as AND2, OR2, XOR2, inverter, D-flip-flop, D-latch, and other gates that are either black boxes for special use or are complex gates such as fulladder and MUX2, which have a direct optimized NCL implementation. The Synopsys design compiler or Cadence Encounter RTL compiler is used in this step. In contrast to [4], here, the designer does not have to specify explicitly the location of registers in the RTL design. Next, the signals are converted to dual-rail, and all Boolean gates are replaced with their NCL implementation. The handshaking signals for the dual-rail registers are then generated and connected to make a functional NCL circuit. The rest of the UNCLE flow is related to optimizing the circuit by performing some optional optimization steps such as latch balancing, relaxation [20], [21], cell merging, and net buffering. An extension of the UNCLE design flow allows a control-driven design style (Balsa style [22], [23]) versus the data-driven style, which can potentially result in a smaller and more energy-efficient NCL circuit.

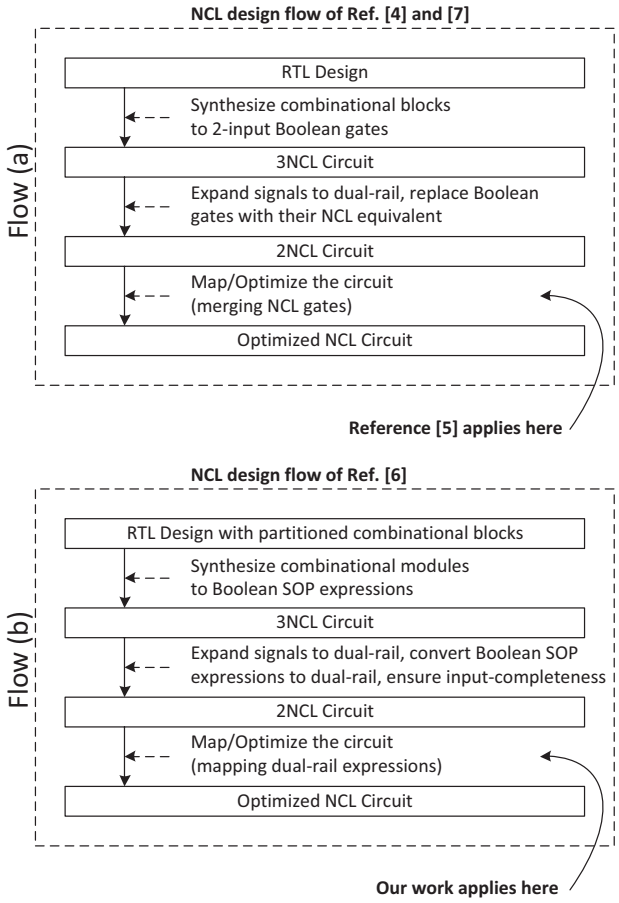

Fig. 1. Automated NCL design flows.

The current NCL design flows can be roughly divided into two main categories. These categories are shown in Fig. 1. The steps shown in both categories are simplified to show the essence of each work. Both design flows start with an RTL design, but the RTL design in Flow (b) requires the combinational blocks to be partitioned into smaller modules (usually having at most four inputs and any number of outputs). This extra constraint is equivalent to the constraint of synthesizing combinational blocks to only two-input Boolean gates in Flow (a). In fact, both of these constraints are imposed by the fan-in restrictions of the standard NCL gate library. If these constraints are not set, the synthesized circuit may contain gates [Flow (a)] or product terms [Flow (b)] that cannot be mapped to NCL gates without breaking them into smaller gates or smaller product terms. In contrast to synchronous circuits, breaking gates or product terms to smaller pieces is not trivial in NCL circuits, since this can potentially introduce gate orphans [5]. Moreover, breaking the large gates or product terms to smaller pieces is usually more expensive in terms of area and delay compared to avoiding them in the first place. For these reasons, all the current NCL design flows (including our algorithm) assume that the synthesized circuits do not contain large gates or large product terms by setting appropriate constraints on the circuit before or during synthesis.

As depicted in Fig. 1, both categories require the initial RTL design to be synthesized. However, in Flow (a) the combinational blocks are synthesized into two-input Boolean gates, while in Flow (b) each combinational module is synthesized into Boolean SOP expressions describing its outputs in terms of its inputs. In other words, the 3NCL circuit resulting from each flow is different since one contains Boolean gates [Flow (a)] and the other contains Boolean SOP expressions [Flow (b)]. As explained previously, although both flows use synthesis tools, Flow (a) requires more sophisticated synthesis tools for synthesis and later for mapping and optimization. The next step in both flows is to expand the single-rail signals to dual-rail signals and replace the Boolean gates with their NCL equivalent [Flow (a)] or convert the Boolean SOP expressions to dual-rail SOP expressions [Flow (b)]. The resultant 2NCL circuits are again different because one contains NCL gates while the other contains dual-rail SOP expressions.

The 2NCL circuits then need to be optimized and mapped before a robust optimized NCL circuit is achieved. Each one of the previous works has developed its own optimization and mapping techniques. In [4], initially the design compiler is used for a constrained minimization of the 2NCL circuit, and then template-based cell merger method is used to further optimize the circuit. In [7], merging adjacent gates is allowed with no fan-out to more complex gates to save area. More systematic and general cell merging and technology mapping techniques are offered in [5], which can be added to Flow (a) to further optimize the final circuit. Similarly, [6] has its own grouping (to be explained later) and mapping algorithm, but as mentioned before, we have proposed an alternative algorithm that can result in a more optimized circuit. Fig. 1 also shows where the algorithm in [5] and ours fit in the existing NCL design flows. Although the algorithm in [5] and our proposed algorithm are only used in the last step of the NCL design flow, it is the most important step, since it determines the efficiency of the final NCL circuit.

Each one of the two NCL design flow categories has its own merits and drawbacks. The first advantage of Flow (b) is its minimum reliance on the expensive sophisticated synthesis CAD tools, while Flow (a) heavily uses them. Although reusing the existing rich set of synchronous CAD tools for designing NCL circuits is beneficial, enabling the NCL design flow to use simpler and cheaper CAD tools makes it more affordable.

Another advantage of Flow (b) is that it allows the designer to have better control over the structure of the final circuit. This control comes by explicitly defining the combinational modules through partitioning the combinational blocks. On the other hand, in Flow (a) the whole combinational block is synthesized into two-input Boolean gates; moreover, the subsequent optimization and mapping steps can potentially remove some of the circuit nodes. Consequently, the resultant circuit will have a less informative structure, which makes the debugging process harder if any problem is encountered later in the design phase. Although partitioning the design to smaller modules can be considered an advantage, it puts more burden on the designer, making Flow (b) more appropriate for custom designs.

Finally, in contrast to Flow (a), Flow (b) takes advantage of the weak conditions of Seitz by only making a minimum

TABLE I
COMPARISON OF THE TWO NCL DESIGN FLOW CATEGORIES

Flow (a)	Flow (b)
Requires more sophisticated synthesis tools	Can work with cheaper and simpler synthesis tools
Provides less control over the final circuit structure	Provides more control over the final circuit structure
Debugging the final circuit is more difficult	Debugging the final circuit is easier
Has area overhead due to not considering the weak conditions of Seitz	Takes advantage of the weak conditions of Seitz to reduce area
Suitable for less custom designs	Suitable for more custom designs

number of outputs of the combinational modules input-complete, which leads to a smaller (and potentially faster) circuit. However, in Flow (a), all the two-input Boolean gates are expanded to their input-complete NCL equivalents without exception, which significantly increases the area. The final area and speed of the synthesized circuits is design-dependent and can be better or worse for each flow. A summary of the advantages and disadvantages of each flow category is shown in Table I. We have designed several circuits using both flows, and the results are compared in Section VI.

C. Contributions of This Paper

This paper addresses the problem of mapping multi-rail logic expressions to an NCL gate library. It particularly focuses on the only other existing multi-rail logic expression mapping algorithm in the literature, which was introduced in [6], and will be called the original algorithm hereafter. The original algorithm will be discussed in detail, and a new mapping algorithm will be proposed that can further reduce the area and delay of the mapped NCL circuits. In contrast to the original algorithm, the new mapping algorithm can target different cost functions or use a subset of NCL gates for mapping. The proposed mapping algorithm is not only helpful for being used as a part of the NCL design automation flow, but it also can be used as a stand-alone tool in assisting the custom NCL design when the derived multi-rail logic expressions become so large that finding the optimal mapping for them becomes difficult.

In summary, this paper makes the following contributions:

1) a new algorithm for mapping multi-rail logic expressions to complex NCL gates is proposed that improves the area and delay of the mapped circuits;
2) the new mapping algorithm can target different cost functions or use only a desired subset of NCL gates for mapping;
3) the new mapping algorithm can be incorporated as part of the NCL design automation flow or be used as a stand-alone tool in a custom NCL design flow to assist mapping large multi-rail logic expressions.

D. Paper Conventions

Assume there is a dual-rail function F expressed as an SOP of certain dual-rail signals. For example, $F^1 = A^1B^1$, and $F^0 = A^0B^0 + A^1B^0 + A^0B^1$. This is an input-complete dual-rail AND function. F^1 corresponds to rail1 of the function F and comprises only one product term, while F^0 corresponds to rail0 of function F and comprises three product terms. F^0 consists of four distinct variables but includes six variables (A^0 and B^0 are repeated). F^1 consists of two distinct variables and includes two variables. For the sake of simplicity, for the rest of this paper, the multi-rail SOP expression will be referred to as the SOP expression, and the product term will be referred to as the term. Assuming that each rail of a multi-rail signal is an independent variable, the SOP expression for F^1 and F^0 can be presented in a simpler form. For example, assuming $A^0 = A, B^0 = B, A^1 = C$, and $B^1 = D$, the SOP expression for F^0 reduces to $AB + BC + AD$, which maps to an NCL THand0 gate, and F^1 reduces to CD, which maps to an NCL TH22 gate. The standard NCL gate library is comprised of 27 gates, as shown in Table II (transistor numbers are for static CMOS implementation [11]). In practice, any SOP expression of four or fewer distinct variables can be directly mapped to one of these 27 NCL gates [3]. This property will be called k-feasibility, where k can be 2, 3, or 4 due to the fan-in limitation of the NCL gate library. In the previous example, F^0 is 4-feasible because it consists of four distinct variables, while F^1 is 2-feasible.

If an SOP expression contains more than four distinct variables, then it must be partitioned into sufficiently small pieces of four or fewer distinct variables before being mapped to NCL gates. This partitioning process is equivalent to dividing the SOP expression into several groups of terms such that each group contains four or fewer distinct variables (i.e., it is k-feasible) and therefore can be directly mapped to an NCL gate. This process is called grouping. As an example, assume F^1 is $A^0B^1C^1+A^0B^1D^1+A^1B^1+B^1C^0+B^1D^0+A^1C^1$. We will see that F^1 can be divided into several k-feasible groups: $A^0B^1C^1+A^1B^1+A^1C^1$, $A^0B^1D^1+B^1C^0$, and B^1D^0, where each group is k-feasible; thus, it can be mapped to an NCL gate. However, F^1 can be alternatively divided into several other k-feasible groups; in other words, grouping is not unique. The question is how to group the terms such that the mapped circuit is optimal in terms of a certain cost function. This paper aims to answer this question through the proposed grouping algorithm. Each k-feasible group is finally mapped to a distinct NCL gate, so group and gate are equivalent terms in this paper. Also, mapping and grouping refer to the same activity in this paper and are interchangeable.

E. Paper Outline

The rest of this paper is organized as follows. In Section II, the original mapping algorithm is explained in detail, and its constraints are then discussed in Section III. Section IV presents our proposed mapping algorithm; and its improvements over the original algorithm are discussed in Section V. In Section VI, the experimental results for each mapping algorithm are presented by running the developed Perl scripts on some typical multi-rail logic expressions and circuit components. Finally, the conclusion is presented in Section VII.

TABLE II
LIBRARY OF THE 27 STANDARD NCL GATES

Threshold Gate	Set Function	No. of Transistors
TH12	$A + B$	6
TH22	AB	12
TH13	$A + B + C$	8
TH23	$AB + AC + BC$	18
TH33	ABC	16
TH23w2	$A + BC$	14
TH33w2	$AB + AC$	14
TH14	$A + B + C + D$	10
TH24	$AB + AC + AD + BC + BD + CD$	26
TH34	$ABC + ABD + ACD + BCD$	24
TH44	$ABCD$	20
TH24w2	$A + BC + BD + CD$	20
TH34w2	$AB + AC + AD + BCD$	22
TH44w2	$ABC + ABD + ACD$	23
TH34w3	$A + BCD$	18
TH44w3	$AB + AC + AD$	16
TH24w22	$A + B + CD$	16
TH34w22	$AB + AC + AD + BC + BD$	22
TH44w22	$AB + ACD + BCD$	22
TH54w22	$ABC + ABD$	18
TH34w32	$A + BC + BD$	17
TH54w32	$AB + ACD$	20
TH44w322	$AB + AC + AD + BC$	20
TH54w322	$AB + AC + BCD$	21
THXOR0	$AB + CD$	20
THAND0	$AB + BC + AD$	19
TH24comp	$AC + BC + AD + BD$	18

II. ORIGINAL MAPPING ALGORITHM

After several synthesis and optimization steps in [6], each combinational module in the initial synchronous RTL design is finally expressed as dual-rail SOP expressions describing its outputs in terms of its inputs. Then, each SOP expression is separately mapped to NCL gates using the grouping algorithm in [6], as shown in Fig. 2. The algorithm starts with a multi-rail SOP expression as input and then groups the product terms of the SOP expression such that an area-efficient implementation is obtained. Taking $F^1 = A^0B^1C^1 + A^0B^1D^1 + A^1B^1 + B^1C^0 + B^1D^0 + A^1C^1$ as an example, the first step in the original grouping algorithm is to remove all four-variable terms from the initial SOP expression, because according to the NCL gate library, these terms cannot be grouped with any other terms to make larger groups and are always realized using TH44 gates. Thus, any four-variable term in the initial SOP expression can be individually regarded as a group and moved to the set of final groups. Since F^1 has no four-variable term, no action is needed. Next, the distinct variables in the SOP expression are counted; if the number is less than 4, the previously grouped terms can be potentially combined with the SOP expression to make 4-feasible groups (this can potentially reduce area and will be discussed later).

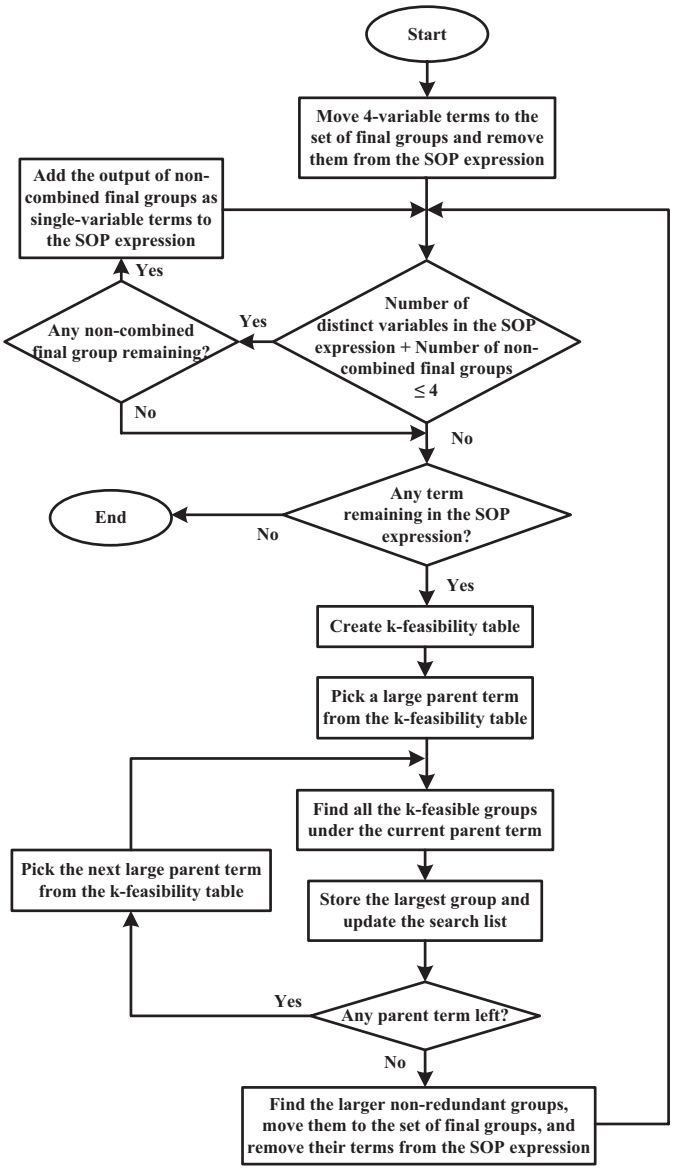

Fig. 2. Original grouping algorithm.

F^1 contains seven distinct variables, so the algorithm proceeds to the next decision box, where it is determined whether the SOP expression is empty or not. The initial SOP expression has not yet been modified, so the algorithm proceeds.

The next step in the original grouping algorithm is to create a k-feasibility table (Table III). Each column in this table corresponds to one of the terms in the SOP expression. This term is called a parent term and, under each parent term in the k-feasibility table, the other terms (same size or smaller) that can be combined with the parent term such that their combination includes four or fewer distinct variables are listed. For example, the first column in Table III corresponds to the parent term $A^0B^1C^1$. Since the combination of $A^0B^1D^1$ with the parent term includes four distinct variables, it is added under the parent term. The rest of the table is built the same way.

Now, starting from the larger parent terms and proceeding to the smaller ones (i.e., first three-variable parent terms, then

TABLE III
k-FEASIBILITY TABLE

$A^0B^1C^1$	$A^0B^1D^1$	A^1B^1	B^1C^0	B^1D^0	A^1C^1
A^0B^1	$A^0B^1C^1$	B^1C^0	A^1B^1	A^1B^1	A^1B^1
A^1B^1	A^1B^1	B^1D^0	B^1D^0	B^1C^0	B^1C^0
B^1C^0	B^1C^0	A^1C^1	A^1C^1	A^1C^1	B^1D^0
B^1D^0	B^1D^0				
A^1C^1					

TABLE IV
k-FEASIBLE GROUPS FOR EACH PARENT TERM

$A^0B^1C^1$	$A^0B^1D^1$	A^1B^1
$A^0B^1C^1 + A^0B^1D^1$	$A^0B^1D^1 + A^1B^1$	$A^1B^1+B^1C^0+B^1D^0$
$A^0B^1C^1+A^1B^1+A^1C^1$	$A^0B^1D^1 + B^1C^0$	$A^1B^1+B^1C^0+A^1C^1$
$A^0B^1C^1 + B^1C^0$	$A^0B^1D^1 + B^1D^0$	$A^1B^1+B^1D^0+A^1C^1$
$A^0B^1C^1 + B^1D^0$		

two-variable parent terms, and finally single-variable parent terms), one parent term is picked at a time, and all the possible k-feasible groups are found by combining the parent term with the other terms listed under it. In Table III, the parent term $A^0B^1C^1$ can be combined with the terms listed under it to form the following k-feasible groups: $\{A^0B^1C^1 + A^0B^1D^1, A^0B^1C^1 + A^1B^1 + A^1C^1, A^0B^1C^1 + B^1C^0$, and $A^0B^1C^1 + B^1D^0\}$. Note that in the original grouping method, when the terms are combined the largest possible groups are formed, and the temporary small groups are discarded. For example, since $A^0B^1C^1 + A^1B^1$ can be combined with A^1C^1 to make the larger group $A^0B^1C^1+A^1B^1+A^1C^1$, the original smaller group is discarded.

Using the same procedure, all the k-feasible groups for the other parent terms are derived. The k-feasible groups for F^1 are shown in Table IV (repeated groups under different parent terms are not shown). In the original grouping method, in order to reduce the runtime, after finding k-feasible groups for each parent term, the largest group is selected, and the rest of the groups are removed. Also, the selected group is added to a "search list" to avoid finding the same group for the other parent terms (this is possible because, for example, both parent terms $A^0B^1C^1$ and $A^0B^1D^1$ can form the same group $A^0B^1C^1 + A^0B^1D^1$). Table V shows the largest group under each parent term. A group is the largest of all groups under a parent term if it has more terms than the other groups. If all groups under a parent term have the same number of terms, the one that has more variables is the largest group. In the original grouping method, if groups of exactly the same size fall under a parent term, the first group is always selected and the rest are discarded. For example, in Table IV, all the groups under parent $A^0B^1D^1$ have the same size, so the first one is selected, and the rest are discarded. It will be shown that this approach of selecting the largest group can potentially result in nonoptimal grouping; the new grouping algorithm addresses this limitation.

The original grouping algorithm then proceeds with choosing the largest non-redundant groups from the set of largest

TABLE V
FINAL k-FEASIBLE GROUPS AFTER FIRST ITERATION

$A^0B^1C^1$	$A^0B^1D^1$	A^1B^1
$A^0B^1C^1+A^1B^1+A^1C^1$	$A^0B^1D^1+A^1B^1$	$A^1B^1+B^1C^0+B^1D^0$

groups found under each parent term, and moving them to the set of final groups. A group is considered non-redundant if it does not share a common term with any other group. Shared terms are not allowed in the set of final groups since they introduce gate orphans and therefore impact the delay-insensitivity of NCL circuits. This issue arises because, when several gates share a common term, there is a possibility that all their outputs transition during the same DATA phase, but in that case only the fastest transition is acknowledged by the output and the other slower transitions are not acknowledged. In the original grouping method, non-redundant groups are found as follows: starting from the first group in the list, a group is selected and compared with the other groups; if it shares a common term with any other group, then the larger group is selected and the smaller group is discarded. This procedureis continued until all the smaller redundant groups are removed. The remaining groups, which are large and non-redundant, are moved to the set of final groups. For example, in Table V, $A^0B^1C^1+A^1B^1+A^1C^1$ is first compared with $A^0B^1D^1 + A^1B^1$ and, because they both share A^1B^1, the smaller group (i.e., $A^0B^1D^1 + A^1B^1$) is removed. Next, $A^0B^1C^1 + A^1B^1 + A^1C^1$ is compared with $A^1B^1 + B^1C^0 + B^1D^0$, and for the same reason, $A^1B^1 + B^1C^0 + B^1D^0$ is removed. At the end, only $A^0B^1C^1 + A^1B^1 + A^1C^1$ remains; and it is moved to the set of final groups. This method of finding the largest non-redundant groups will be proven to be inefficient, and its limitations will be addressed in the new grouping method.

At the end of the first iteration in the original grouping algorithm, the selected groups are moved to the set of final groups, and their terms are removed from the initial SOP expression. In the previous example, the selected group (i.e., $A^0B^1C^1 + A^1B^1 + A^1C^1$) is removed from the initial SOP expression, reducing it to $A^0B^1D^1 + B^1C^0 + B^1D^0$.

The same procedure is then repeated for the new SOP expression until all the terms in the expression are grouped. For the previous example, at the end of the second iteration, $A^0B^1D^1 + B^1C^0$ is selected and moved to the set of final groups. After removing this group from the SOP expression, the initial SOP expression reduces to B^1D^0. At the beginning of the third iteration, the condition in the first decision box in the grouping algorithm is satisfied because only two variables remain in the SOP expression (i.e., B^1 and D^0). This situation normally means that the last group will be smaller, but sometimes it is possible to add more variables to the remaining SOP expression to make the last group larger and potentially reduce the total number of final groups. The current set of final groups can be used for this purpose because the final groups eventually need to be ORed together in order to form the output. Therefore, the outputs of the current set of final

TABLE VI
FINAL GROUPS

Final Groups
$T1 = A^0B^1C^1 + A^1B^1 + A^1C^1$
$T2 = A^0B^1D^1 + B^1C^0$
$F^1 = B^1D^0 + T1 + T2$

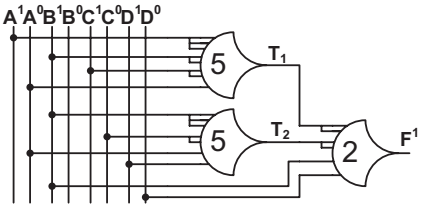

Fig. 3. NCL implementation.

groups are added as single-variable terms to the current SOP expression at this step. In the previous example, two groups already exist in the set of final groups. We will call the output of the first group T1 and the output of the second group T2. After adding T1 and T2 to the SOP expression, the new expression becomes $B^1D^0 + T1 + T2$. At the end of the third iteration, $B^1D^0 + T1 + T2$ is selected as a final group and is removed from the SOP expression; therefore, the SOP expression becomes empty, and the grouping algorithm ends. Table VI shows the final groups and Fig. 3 shows its NCL implementation.

If T1 and T2 were not added to the SOP expression, then B^1D^0 would be selected as the third final group. Assuming the output of this group was called T3, another group (and also another iteration) would be required to OR the output of the final groups (i.e., $F^1 = T1 + T2 + T3$). Therefore, adding the output of the final groups to the SOP expression when it has less than four variables can potentially reduce the number of final groups and consequently produce a more area-efficient circuit. Although this approach appears helpful, in some cases it can increase both the area and the delay of a circuit. This problem can be resolved by adding a postprocessing step as will be discussed in the new grouping algorithm.

III. ORIGINAL MAPPING ALGORITHM CONSTRAINTS

Several potential constraints of the original grouping algorithm were introduced in the previous section. In this section, these constraints are discussed in more detail and some examples are provided. In the next section, our proposed grouping algorithm is introduced, and these constraints are addressed.

The original grouping method only targets area as the cost function and tries to minimize area by finding larger groups and decreasing the number of final groups. It also assumes that all 27 NCL gates are available for grouping. However, in practice, some NCL gates may not be available, or a designer could be interested in targeting delay as the cost function by using a certain set of low-latency NCL gates for mapping. The original grouping method cannot provide such a degree of flexibility. Moreover, the original grouping method usually requires several iterations to find the final groups (multi-pass grouping). Based on the original grouping algorithm shown in Fig. 2, all the steps, from creating a k-feasibility table to combining terms with the parents, finding the largest group under each parent, and finally finding the largest non-redundant groups, must be performed every time the main loop is executed. This execution is computationally expensive and could be alleviated by engaging in less computation in the main loop.

As explained previously, in the original grouping method, after finding all k-feasible groups under a parent term, the largest group is selected and the rest are discarded. However, if the k-feasible groups under a parent term have exactly the same size, then the first group is selected as the largest group. This approach of selecting the largest group can sometimes result in an inefficient grouping. For example, assume $F^1 = A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0 + A^0D^0 + B^0C^1D^1$. In this case, the 4-feasible groups under parent A^0B^1 would be $\{A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0$, and $A^0B^1 + A^0C^0 + B^1C^0 + A^0D^0\}$. Now, if the first group is selected, the final grouping would be {T1 = $A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0$ (TH44w322), T2 = $B^0C^1D^1$ (TH33), and $F^1 = A^0D^0 + T1 + T2$ (TH24w22)}. However, if the second group under parent A^0B^1 is selected as the largest group, the new grouping would be {T1 = $A^0B^1 + A^0C^0 + B^1C^0 + A^0D^0$ (TH44w322), T2 = $B^0C^1D^1 + A^0D^1$ (TH54w32), and $F^1 = T1 + T2$ (TH12)}. Based on Table II, this grouping saves six transistors compared to the first grouping and is also faster (see Section V).

The original grouping method is also not efficient in terms of finding the largest non-redundant groups. As discussed in the previous section, starting from the first group, each group is compared with the other groups, and if they share common terms, the largest one is selected, and the other one is discarded. For example, assume $F^1 = B^1C^0D^0 + C^1D^0 + A^0B^1 + A^0C^1 + A^0D^1 + B^0C^1$. The set of the largest groups after the first iteration will contain $\{B^1C^0D^0 + C^1D^0$, $C^1D^0 + A^0B^1 + A^0C^1$, $A^0B^1 + A^0C^1 + A^0D^1$, $A^0C^1 + A^0D^1 + B^0C^1\}$. Using the original method to find the largest non-redundant groups results in only $C^1D^0 + A^0B^1 + A^0C^1$ remaining at the end. However, a more careful examination reveals that selecting two other large non-redundant groups, which the original grouping method would never select, is indeed a better choice (i.e., $A^0B^1 + A^0C^1 + A^0D^1$ and $B^1C^0D^0 + C^1D^0$). This alternative choice not only decreases the algorithm runtime but also can result in a smaller and faster circuit. In fact, if the first choice is used, the final groups would be {T1 = $A^0B^1 + A^0C^1 + C^1D^0$ (TH and 0), T2 = $A^0D^1 + B^0C^1$ (TH xor 0), T3 = $B^1C^0D^0 + T1$ (TH34w3), $F^1 = T2 + T3$ (TH12)}, but if the second choice is used, the final groups would be {T1 = $A^0B^1 + A^0C^1 + A^0D^1$ (TH44w3), T2 = $B^1C^0D^0 + C^1D^0$ (TH54w32), $F^1 = B^0C^1 + T1 + T2$ (TH24w22)}. The second grouping saves 11 transistors and is faster (see Section V).

Finally, the approach of adding the final groups as single-variable terms to the initial SOP expression to make larger groups is not always helpful. In fact, this approach can potentially result in a larger circuit or increase the logic levels unnecessarily and consequently make the final circuit slower. This technique is usually beneficial only if it can

reduce the total number of final groups. For example, assume $F^1 = A^0B^1C^0 + A^0B^0C^1 + A^1B^1C^0 + A^1B^0C^1 + B^1C^0D^1$. Using the original grouping method, the final groups would be {T1 = $A^0B^1C^0 + A^1B^1C^0$ (TH54w22), T2 = $A^0B^0C^1 + A^1B^0C^1$ (TH54w22), T3 = $B^1C^0D^1$+ T1 (TH34w3), F^1 = T2 + T3 (TH12)}. In this example, combining T1 with $B^1C^0D^1$ cannot decrease the number of final groups, but it adds to the delay of the circuit. In fact, a grouping in which T3 = $B^1C^0D^1$ (TH33) and F^1 = T1 + T2 + T3 (TH13) maintains the same number of transistors but cuts the circuit delay almost in half (see Section V).

IV. Proposed Mapping Algorithm

The standard NCL gate library is comprised of 27 gates, each of which has several properties. Some of these properties are technology-independent, such as the set function and the number of transistors, while some are technology-dependent, such as area and delay. The main idea of the new grouping algorithm is to sort the list of NCL gates based on a cost function prior to grouping. At the time of grouping, the gates occupying a higher position in the sorted list are considered more important and will have a higher priority in grouping. Moreover, in contrast to the original grouping method, the list of NCL gates is no longer required to include all 27 gates as long as it contains enough gates to cover any given SOP expression. This usually means that the gate list must contain at least all of the NCL AND (THnn) and OR (TH1n) gates.

The first step in using the proposed grouping algorithm is to select a set of NCL gates to which the SOP expressions are to be mapped. The selected NCL gates are then sorted on the basis of a cost function, and each gate is assigned a priority number accordingly. At the time of grouping, this priority number is used to determine which groups are most desirable. The cost function is usually area or delay. For example, if the cost function is area, the gates that cover more terms with less area will have a higher priority. In other words, let F be a multi-rail SOP expression. Assume that function F can be implemented (denoted by $<\bullet$) by the set of NCL gates $\{H_0, H_1, \ldots, H_n\}$

$$\{H_0, H_1, \ldots, H_n\} <\bullet\ F.$$

If there exists a single NCL gate K that implements the same function

$$K <\bullet\ F.$$

Then, K is said to have a higher priority than any NCL gate in the set of $\{H_0, H_1, \ldots, H_n\}$ if

$$\text{Area}(K) \leq \sum_{i=0}^{n} \text{Area}(H_i).$$

As an example, consider $F(A, B, C, D) = AB + ACD + BCD$. F can be implemented using a single gate TH44w22, or it can be implemented using several other gates such as $\{H = AB$ (TH22), $H_1 = ACD + BCD$ (TH54w22), $H_2 = H + H_1$ (TH12)}. Since TH44w22 has less area (no. of transistors) compared to the sum of the area for TH54w22, TH22, and TH12 it has a higher priority than all of them.

TABLE VII
Priority Table

Priority Level	NCL Threshold Gate	Set Function	3v and 2v Terms Covered	Variables Covered
1	TH24	$AB + AC + AD + BC + BD + CD$	6	12
2	TH34w22	$AB + AC + AD + BC + BD$	5	10
3	TH34	$ABC + ABD + ACD + BCD$	4	12
4	TH34w2	$BCD + AB + AC + AD$	4	9
5	TH44w322 TH24comp	$AB + AC + AD + BC$ $AC + BC + AD + BD$	4	8
6	TH44w2	$ABC + ABD + ACD$	3	9
7	TH44w22	$ACD + BCD + AB$	3	8
8	TH54w322	$BCD + AB + AC$	3	7
9	TH24w2	$BC + BD + CD + A$	3	7
10	TH23 TH44w3 THand0	$AB + AC + BC$ $AB + AC + AD$ $AB + BC + AD$	3	6
11	TH54w22	$ABC + ABD$	2	6
12	TH54w32	$ACD + AB$	2	5
13	TH34w32	$BC + BD + A$	2	5
14	TH33w2 THxor0	$AB + AC$ $AB + CD$	2	4
15	TH34w3	$BCD + A$	1	4
16	TH33	ABC	1	3
17	TH24w22	$CD + A + B$	1	4
18	TH23w2	$BC + A$	1	3
19	TH22	AB	1	2

For the rest of this paper, for the sake of comparison with the original grouping method, let us assume that the cost function is area and that all of the NCL gates are available for grouping. A typical priority table is shown in Table VII, in which the gates are sorted based on how many three-variable and two-variable terms they cover. If some gates cover the same number of terms, the one that covers more variables is assigned a higher priority; if they also cover the same number of variables, then they are assigned the same priority number. Note that the NCL OR gates (TH1n) are missing from this table because they are used in the last step of the proposed grouping algorithm when the final groups are ORed. This step will be discussed later in more detail.

The proposed grouping algorithm is shown in Fig. 4. In summary, the algorithm begins with moving all the four-variable terms to the set of final groups, just as in the original grouping method. Next, all k-feasible groups are found by using a k-feasibility table and combining all parent terms with the other terms, as discussed in the original grouping method. The k-feasible groups are then arranged into priority levels according to Table VII. Next, starting from the highest non-empty priority level, the non-redundant groups in each priority level are found, their terms are removed from all the other groups, and the priority levels are updated accordingly. This procedure continues until all priority levels become empty.

Let us take the same SOP expression that was used to explain the original grouping algorithm as an example. Assume $F^1 = A^0B^1C^1 + A^0B^1D^1 + A^1B^1 + B^1C^0 +$

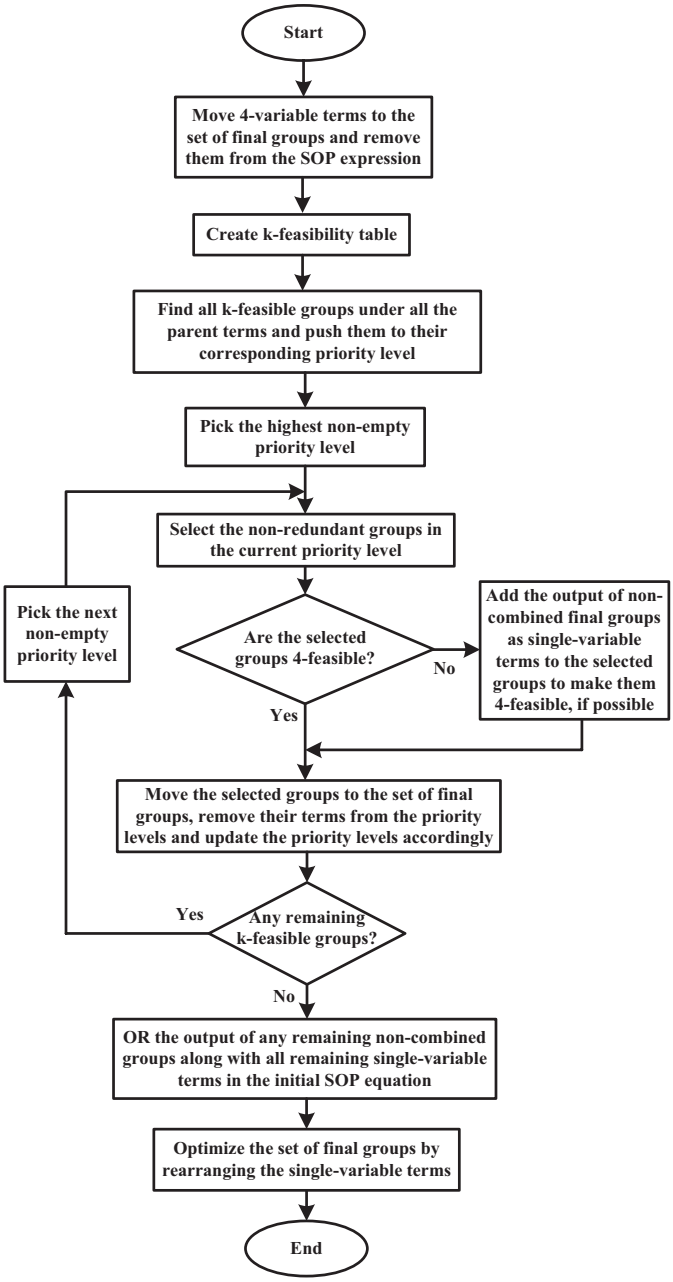

Fig. 4. Proposed grouping algorithm.

$B^1D^0 + A^1C^1$. Based on the algorithm shown in Fig. 4, the first step is to remove all four-variable terms from the SOP expression and move them to the set of final groups for the same reason that they were removed in the original grouping method. F^1 contains no four-variable terms, so this step is skipped. All k-feasible groups are then found by using the k-feasibility table, as discussed in the original grouping method. Table VIII shows the result. A major difference exists between the k-feasible groups found with the original grouping method and the ones found with the proposed grouping method. As mentioned previously, with the original grouping method, only larger k-feasible groups are stored, and the smaller temporary k-feasible groups are discarded. In the proposed grouping algorithm, however, all possible combinations of terms are stored. For example, for the parent A^1B^1 in Table VIII, $A^1B^1 + B^1C^0 + B^1D^0$, $A^1B^1 + B^1C^0$, and A^1B^1 are all stored. This enables the proposed grouping algorithm to find all the final groups in one iteration (one-pass grouping) as opposed to the original grouping algorithm that requires several iterations (see Section III). Finding all k-feasible groups does not require more processing compared to the original grouping method since the process of finding larger k-feasible groups inherently includes finding smaller ones.

The next step is to push the groups into their corresponding priority levels, as demonstrated in Table IX. In this table, under each priority level, one or more groups from Table VIII are listed. The empty priority levels are not shown. The algorithm then picks one priority level at a time, starting from the higher non-empty priority levels, and then selects the non-redundant groups in each priority level. In Table IX, the highest priority level is 8, and this level includes only one group (i.e., $A^0B^1C^1 + A^1B^1 + A^1C^1$), which is selected as a non-redundant group. The selected groups are then examined to determine whether they are 4-feasible (i.e., comprised of four distinct variables); if they are not, the output of the previously found final groups are added to them as single-variable terms to make them 4-feasible, if possible (provided that the new group is still implementable in the chosen NCL gate library). This approach is almost similar to the one used in the original grouping method. This step may not be required if the cost function is delay and can be skipped without causing problems for the whole grouping algorithm (in contrast to the original grouping method which will fail). Since the selected group in the priority level 8 is already 4-feasible, no action is required.

The selected groups are finally moved to the set of final groups, their terms are removed from the priority levels, and the priority levels are updated accordingly. For example, after removing the terms of $A^0B^1C^1 + A^1B^1 + A^1C^1$ from the priority levels, $A^1B^1 + B^1C^0 + B^1D^0$, which already belongs to the priority level 10, reduces to $B^1C^0 + B^1D^0$, which now belongs to the priority level 14 according to Table VII. The updated priority levels are shown in Table X. This procedure is repeated for the other priority levels until all the product terms are grouped and the priority levels become empty. In Table X, the next highest priority level is 12, which contains two groups. Because these two groups share a common term, they are dependent, and one of them must be removed. In this example, it does not matter which one is removed (as will be explained later), so the first group is selected and the other is removed. The selected group is then moved to the set of final groups, and its terms are removed from the priority levels. This leaves only B^1D^0, which can be combined with the other final groups as shown in Table XI. The corresponding NCL implementation is shown in Fig. 5. For this example, the original grouping method and the proposed one produce the same grouping result, but this is not always the case as explained in the next section.

The proposed grouping method always preserves delay-insensitivity of the initial dual-rail function. In order to prove it, one must prove that the proposed grouping method preserves input-completeness and observability.

TABLE VIII
k-FEASIBLE GROUPS

$A^0B^1C^1$	$A^0B^1D^1$	A^1B^1	B^1C^0	B^1D^0	A^1C^1
$A^0B^1C^1 + A^0B^1D^1$	$A^0B^1D^1 + A^1B^1$	$A^1B^1 + B^1C^0 + B^1D^0$	$B^1C^0 + B^1D^0$	B^1D^0	A^1C^1
$A^0B^1C^1 + A^1B^1 + A^1C^1$	$A^0B^1D^1 + B^1C^0$	$A^1B^1 + B^1C^0 + A^1C^1$	B^1C^0		
$A^0B^1C^1 + B^1C^0$	$A^0B^1C^1 + B^1D^0$	$A^1B^1 + B^1D^0 + A^1C^1$			
$A^0B^1C^1 + B^1D^0$	$A^0B^1D^1$	$A^1B^1 + B^1C^0$			
$A^0B^1C^1$		$A^1B^1 + B^1D^0$			
		$A^1B^1 + A^1C^1$			
		A^1B^1			

TABLE IX
k-FEASIBLE GROUPS PUSHED TO THEIR CORRESPONDING PRIORITY LEVELS

8	**10**	**11**	**12**	**14**	**16**	**19**
$A^0B^1C^1 + A^1B^1 + A^1C^1$	$A^1B^1 + B^1C^0 + B^1D^0$	$A^0B^1C^1 + A^0B^1D^1$	$A^0B^1C^1 + B^1C^0$	$A^1B^1 + B^1C^0$	$A^0B^1C^1$	A^1B^1
	$A^1B^1 + B^1C^0 + A^1C^1$		$A^0B^1C^1 + B^1D^0$	$A^1B^1 + B^1D^0$	$A^0B^1D^1$	B^1C^0
	$A^1B^1 + B^1D^0 + A^1C^1$		$A^0B^1D^1 + A^1B^1$	$A^1B^1 + A^1C^1$		B^1D^0
			$A^0B^1D^1 + B^1C^0$	$B^1C^0 + B^1D^0$		A^1C^1
			$A^0B^1D^1 + B^1D^0$			

TABLE X
AFTER REMOVING THE TERMS OF THE FIRST SELECTED GROUP AND UPDATING THE PRIORITY LEVELS

12	**14**	**16**	**19**
$A^0B^1D^1 + B^1C^0$	$B^1C^0 + B^1D^0$	$A^0B^1D^1$	B^1C^0
$A^0B^1D^1 + B^1D^0$			B^1D^0

TABLE XI
FINAL GROUPS

Final Groups
T1 = $A^0B^1C^1 + A^1B^1 + A^1C^1$
T2 = $A^0B^1D^1 + B^1C^0$
F^1 = B^1D^0+ T1 + T2

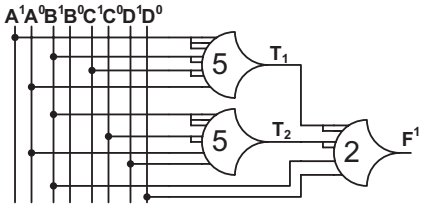

Fig. 5. NCL implementation.

Theorem 1: Grouping preserves input-completeness.

Proof: Let dual-rail function F be input-complete with regard to a certain variable X. Hence, all product terms of F (both F^1 and F^0) include variable X (either X^1 or X^0) [3]. Grouping is the process of combining (Boolean OR) the product terms into k-feasible groups and then ORing these groups to produce the output. Since grouping does not modify the product terms, each term will still contain variable X; therefore, F will still be input-complete with respect to X. ■

Theorem 2: Grouping preserves observability.

Proof: A circuit is observable when it does not include any gate orphans. In other words, any transition on the output of a gate must be followed by a transition on a primary output. Since grouping only combines the product terms into k-feasible groups and then ORs them to produce the final output, an assertion on the output of a gate will pass through the network of OR gates and will assert a primary output. Therefore, the output of all the gates is observable. ■

V. IMPROVEMENTS OVER THE ORIGINAL MAPPING ALGORITHM

Several advantages of the proposed grouping algorithm, such as the ability to use different cost functions, the ability to map to a subset of NCL gates, and one-pass grouping, were pointed out in the previous section. In this section, the details of some of the previously mentioned steps are explained, and it is shown how these steps improve the overall mapping process.

With the proposed grouping method, the non-redundant groups under a priority level are selected efficiently. Finding the maximum number of non-redundant groups is a special case of the *maximal independent set* problem [24]. This problem is NP-complete, so it is unlikely that it can be solved in polynomial time; however, since the number of groups in each priority level is always small (due to the small size of each combinational module, see Section I-B), the runtime is always small. Many algorithms have been proposed for this problem in the literature [25], [26], but for our special case we use a simple approach as explained in Fig. 6. In contrast to the approach that is used in the original grouping method, this approach always finds the maximum number of non-redundant groups in a given set of groups. In Fig. 6, each circle represents a term, and each enclosed area represents a group in the current priority level. These groups are named K1 through K5. In this example, each group covers four terms, and some of the groups

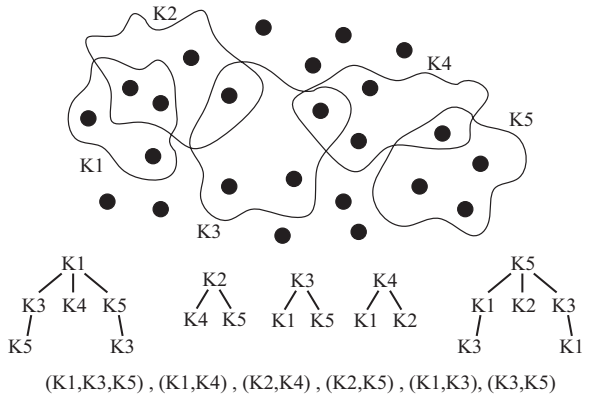

Fig. 6. Dependency trees.

have common terms, and therefore are redundant. The terms that are not covered by any of the groups do not belong to the current priority level. In order to find the maximum number of non-redundant groups, a dependency tree must be made for each group. The following steps explain how to make a dependency tree for a group. First, a group is selected as the root (e.g., K1); then, all the groups that share a common term with the root are removed (K2), and the rest of the groups are added as branches to the root (K3, K4, K5). Then, one of the branches is selected as the new root (e.g., K3), and the other branches are added as branches to the new root if they share no common term with the new root (only K5); otherwise, they are removed (K4). This procedure, which is the same for all other branches, continues until all the branches are removed. The same procedure is followed for the other groups until a dependency tree is created for each group. The dependency trees for the previous example are shown in Fig. 6. Each path (starting from the root to the bottommost branch) in a dependency tree represents a set of non-redundant groups. For example, in the first tree of Fig. 6, there are two sets of non-redundant groups, i.e., (K1, K3, K5) and (K1, K4). The order of groups is not important, so (K1, K5, K3) is considered the same as (K1, K3, K5) and is ignored. The other sets of non-redundant groups are shown in Fig. 6. A set of non-redundant groups that includes more groups is clearly more desirable. In this example, (K1, K3, K5) offers the maximum number of non-redundant groups.

The dependency tree method does not always return a unique solution. Assume a scenario in which the dependency tree method returns multiple sets of non-redundant groups, each having the same number of groups. In this case, the proposed grouping method will try all the possible scenarios recursively in order to find the most optimal grouping. For example, in Table X under priority level 12, there are two dependent groups of the same size. One scenario could be to select $A^0B^1D^1 + B^1C^0$ as the final group and proceed with the grouping, and the other scenario could be to select $A^0B^1D^1+B^1D^0$. In practice, the new grouping method would consider both scenarios before selecting the optimal grouping. Technically, considering all the scenarios will result in a tree that must be traversed in order to find the most optimal grouping. A *depth-first search* approach is used to traverse the resultant tree, which takes $O(|V| + |E|)$ time, where V and E are the number of vertices and edges of the tree, respectively [27].

Algorithm 1 Single-Variable Term Relocation

```
OptimizeMapping (M)
1  S ={gates with single-variable terms in their set function}
2  for (level L = 1 to last level in M) do
3      for (each gate g_i in level L) do
4          if (g_i ∈ S) then
5              move single-variable terms of g_i to level L + 1
6              if (cost(new M) < cost(old M)) then
7                  save the new M
8              else
9                  revert to the old M
```

Finally, the process of using the output of the current final groups as single-variable terms to make larger groups is corrected in the proposed grouping algorithm, by adding an extra optimization step. As mentioned previously, this approach is usually helpful only when it can reduce the number of final groups. Moreover, if the cost function is delay, combining the final groups can be undesirable even if it reduces the number of final groups. The added optimization step in Fig. 4 will explore relocating the single-variable terms in the set of final groups (without increasing the logic levels) to find the optimal grouping. This optimization step is shown in Algorithm 1, where M is the mapping before optimization. In this step, starting from the first logic level, each logic level is searched to find the groups that contain a single-variable term. The standard NCL gate library contains eight gates with this property: $A + BCD$ (TH34w3), $A + BC + BD$ (TH34w32), $A + BC$ (Th23w2), $A + BC + BD + CD$ (TH24w2), $A+B+CD$ (TH24w22), $A+B$ (TH12), $A+B+C$ (TH13), and $A+B+C+D$ (TH14). If any of the above groups exists in the current logic level, the single-variable terms are moved to the next logic level, and the area/delay of the new mapping is calculated and compared to the previous one in order to find the best solution. The running time is a linear function of the number of gates in each mapping.

VI. Experimental Results

In order to compare the original and the proposed grouping methods, both methods were implemented using the Perl programming language.

The developed Perl scripts were then tested on several SOP expressions and circuit components. Table XII shows the results of running the developed scripts on some of the examples used in this paper. For each example, the groups resulting from both grouping methods are shown. Moreover, the corresponding number of transistors (T), area, delay, and runtime are also compared. Both grouping scripts were run on an Intel Core 2 Duo 2.4-GHz machine with 4 GB RAM, and the area and delay information came from an NCL physical cell library realized in a 1.8 V 0.18-μm TSMC CMOS process. Table XII shows that the new grouping method decreases the area and delay of the mapped SOP expressions, while the runtime may increase based on the SOP expression. Although the new grouping method enables one-pass grouping and

TABLE XII
COMPARISON OF THE ORIGINAL AND PROPOSED GROUPING METHODS

	Resulted Groups	# T	Area [μm^2]	Delay [ns]	Runtime [ms]
1	F = $A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0 + A^0D^0 + B^0C^1D^1$				
Original	X = $A^0B^1 + A^0C^0 + A^0D^1 + B^1C^0$ (TH44w322) Y = $B^0C^1D^1$ (TH33) F = $A^0D^0 + X + Y$ (TH24w22)	52	271	660	13.5
Proposed	X = $A^0B^1 + A^0C^0 + A^0D^0 + B^1C^0$ (TH44w322) Y = $A^0D^1 + B^0C^1D^1$ (TH54w32) F = $X + Y$ (TH12)	46	254	434	13.8
2	F = $B^1C^0D^0 + C^1D^0 + A^0B^1 + A^0C^1 + A^0D^1 + B^0C^1$				
Original	X = $A^0B^1 + A^0C^1 + C^1D^0$ (THand0) Y = $A^0D^1 + B^0C^1$ (THxor0) Z = $B^1C^0D^0 + X$ (TH34w3) F = $Y + Z$ (TH12)	63	350	734	13.9
Proposed	X = $A^0B^1 + A^0C^1 + A^0D^1$ (TH44w3) Y = $B^1C^0D^0 + C^1D^0$(TH54w32) Z = $B^0C^1 + X + Y$ (TH24w22)	52	293	704	29.7
3	F = $A^0B^1C^0 + A^0B^0C^1 + A^1B^1C^0 + A^1B^0C^1 + B^1C^0D^1$				
Original	X = $A^0B^1C^0 + A^1B^1C^0$ (TH54w22) Y = $A^0B^0C^1 + A^1B^0C^1$ (TH54w22) Z = $B^1C^0D^1 + X$ (TH34w3) F = $Y + Z$ (TH12)	60	368	739	5.6
Proposed	X = $A^0B^1C^0 + A^1B^1C^0$ (TH54w22) Y = $A^0B^0C^1 + A^1B^0C^1$ (TH54w22) Z = $B^1C^0D^1$ (TH33) F = $X + Y + Z$ (TH13)	60	352	447	11.4

TABLE XIII
MAPPING CIRCUIT COMPONENTS USING THE ORIGINAL AND PROPOSED GROUPING METHODS

Component	Grouping Method	Area [μm^2]	Delay [ns]	Runtime [ms]
Q33MUL	Original	1021	597	79
	Proposed	977	363	65
Q322ADD	Original	1665	822	94
	Proposed	1626	534	80
FLAGS	Original	1581	781	26
	Proposed	1543	493	87
BCDAD	Original	1498	712	21
	Proposed	1498	712	38
LOGIC	Original	5899	415	296
	Proposed	5313	415	486

TABLE XIV
MAPPING CIRCUIT COMPONENTS TO A RESTRICTED SUBSET OF NCL GATES USING THE PROPOSED GROUPING METHOD

Component	Grouping Method	Area [μm^2]	Delay [ns]	Runtime [ms]
Q33MUL	All gates	977	363	65
	Limited gates	1061	415	55
Q322ADD	All gates	1626	534	80
	Limited gates	1979	1338	69
FLAGS	All gates	1543	493	87
	Limited gates	1906	1882	96
BCDAD	All gates	1498	712	38
	Limited gates	1680	841	35
LOGIC	All gates	5313	415	486
	Limited gates	5899	415	401

decreases the processing time in the main loop, as opposed to the original grouping method, the extra optimization steps that are missing in the original grouping method (see Section V) can sometimes increase the total runtime.

In order to observe the benefits of the proposed grouping method at the circuit level, this method was used to design several circuit components as listed in Table XIII. In this table, the first two components are used in a quad-rail NCL multiplier [28], and the last three are used in an NCL 8051 ALU [29]. The experimental results show up to10% area reduction in the LOGIC component and up to a 39% delay improvement in the Q33MUL component.

Table XIV shows the result of mapping the circuit components in Table XIII using the new grouping method when only a restricted subset of NCL gates is available. As explained before, this is not possible in the original grouping method. In this experiment, half of the gates in the standard NCL gate library were removed, and grouping was performed with the other half. The results show that the mapped circuits have more area and delay compared to the case with all the NCL gates available, since less optimization is possible with a restricted subset of gates.

Finally, although the proposed grouping method cannot be directly compared to the other previous works

TABLE XV
COMPARISON OF THE TWO NCL DESIGN FLOW CATEGORIES

Circuit	Flow (a)			Flow (b)		
	#i/#o/#g	Area [μm^2]	Delay [ns]	#i/#o/#g	Area [μm^2]	Delay [ns]
dec2×4	2/4/14	1294	632	2/4/8	529	316
dec3×8	3/8/28	2588	632	3/8/20	915	493
logic4	8/12/24	2426	316	8/12/24	1739	236
logic8	16/24/48	4853	316	16/24/48	3478	236
par4	4/1/6	711	710	4/1/6	711	473
par8	8/1/14	1659	1656	8/1/14	1659	710
comp4	8/3/48	4593	2765	8/3/66	5824	2200
comp8	16/3/104	9978	5294	16/3/146	12 968	4818
mul4×4	8/8/100	9933	3338	8/8/128	11 476	3342
mul8×8	16/16/418	41 641	8279	16/16/640	57 310	8122

(see Section I-B), an indirect comparison is possible through incorporating the new grouping algorithm in a design flow such as Flow (b). For this purpose, we added our grouping algorithm to the NCL design Flow (b) and compared it against the NCL design Flow (a) by synthesizing several circuits listed in Table XV. In Flow (a), the UNCLE toolset [7] is used to convert synchronous circuits to NCL, and then the resultant circuits are further optimized using the ATN_OPT toolset [5]. Both these tools (i.e., UNCLE and ATN_OPT) and the detailed instructions on how to use them are available online for public access at the time of writing this paper. In Flow (b), we used the methods described in [6] to partition a synchronous circuit to smaller modules and derive the optimal input-complete dual-rail output SOP expressions, which are then mapped to NCL gates using our proposed grouping method to build the final circuit. In Table XV, several circuits are listed. These circuits include decoders (2-to-4 line and 3-to-8 line), logic blocks (4- and 8-b), parity checkers (4- and 8-b), comparators (4- and 8-b), and multipliers (4×4 and 8×8). For each circuit, the corresponding number of inputs (#i), outputs (#o), and NCL gates (#g) is also listed. Based on the results, the area and delay advantage for each flow is design-dependent and can be better or worse. Flow (b) particularly works better for the circuits that have more outputs than inputs, so more area reduction becomes possible through making a restricted set of outputs input-complete (i.e., using weak conditions of Seitz). On the other hand, Flow (a) works better when more high-level components are inferred at synthesis time. For example, in the case of the multipliers, since they are mainly synthesized to full-adders, the UNCLE tool replaces these components with the manually optimized NCL full-adders (when converting the 3NCL circuit to 2NCL circuit), which results in much better area as compared to Flow (b). At this time, Flow (b) does not take advantage of hand-optimized NCL components; however, this feature can be incorporated into the flow in future versions.

VII. CONCLUSION

A new mapping algorithm was proposed and compared with the original one for mapping multi-rail logic expressions to the NCL gate library. Both mapping algorithms were implemented in the Perl programming language and tested on some multi-rail logic expressions and circuit components. The proposed mapping algorithm was shown to outperform the original one in terms of area and delay reduction. We showed that the new mapping algorithm could result in up to 10% area reduction and 39% delay reduction. Although the new mapping algorithm performs mapping in one pass requiring less computation in the main loop, its average runtime is higher compared to the original method because the new mapping algorithm incorporates several extra optimization steps. Also, in contrast to the original mapping algorithm, the new mapping algorithm can map a multi-rail expression to a restricted subset of NCL gates and can target any cost function.

REFERENCES

[1] K. M. Fant and S. A. Brandt, "NULL convention logic: A complete and consistent logic for asynchronous digital circuit synthesis," in *Proc. Int. Conf. Appl. Specific Syst., Archit. Process.*, Aug. 1996, pp. 261–273.

[2] K. M. Fant, *Logically Determined Design: Clockless System Design with NULL Convention Logic*. New York: Wiley, 2005.

[3] S. C. Smith and J. Di, "Designing asynchronous circuits using NULL convention logic (NCL)," in *Synthesis Lectures on Digital Circuits and Systems*. San Mateo, CA: Morgan & Claypool, 2009.

[4] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *Proc. 6th Int. Symp. Adv. Res. Asynchron. Circuits Syst.*, Apr. 2000, pp. 114–125.

[5] C. Jeong and S. M. Nowick, "Technology mapping and cell merger for asynchronous threshold networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 4, pp. 659–672, Apr. 2008.

[6] B. Bhaskaran, "Automated synthesis and cycle reduction optimization for asynchronous NULL convention circuits using industry-standard CAD tools," Ph.D. dissertation, Dept. Comp. Eng., Univ. Missouri - Rolla, Rolla, 2007.

[7] R. Reese, S. C. Smith, and M. A. Thornton, "Uncle—an RTL approach to asynchronous design," in *Proc. 18th IEEE Int. Symp. Adv. Res. Asynchron. Circuits Syst.*, May 2012, pp. 65–72.

[8] J. McCardle and D. Chester, "Measuring an asynchronous processor's power and noise," in *Proc. Synopsys Users Group Conf.*, 2001, pp. 66–70.

[9] S. C. Smith, R. F. DeMara, J. S. Yuan, D. Ferguson, and D. Lamb, "Optimization of NULL convention self-timed circuits," *Integr. VLSI J.*, vol. 37, no. 3, pp. 135–165, Aug. 2004.

[10] T. Verhoeff, "Delay-insensitive codes—an overview," *Distrib. Comput.*, vol. 3, no. 1, pp. 1–8, 1988.

[11] G. E. Sobelman and K. Fant, "CMOS circuit design of threshold gates with hysteresis," in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 2. Jun. 1998, pp. 61–64.

[12] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980, pp. 218–262.

[13] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, "Checking delay-insensitivity: 104 gates and beyond," in *Proc. 8th Int. Symp. Asynchron. Circuits Syst.*, Apr. 2002, pp. 149–157.

[14] A. J. Martin, "Programming in VLSI," in *Development in Concurrency and Communication*. Reading, MA: Addison-Wesley, 1990, pp. 1–64.

[15] K. V. Berkel, "Beware the isochronic fork," *Integr. VLSI J.*, vol. 13, no. 2, pp. 103–128, Jun. 1992.

[16] I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of Boolean functions as self-timed circuits," *IEEE Trans. Comput.*, vol. 41, no. 1, pp. 2–11, Jan. 1992.

[17] M. Shams, J. C. Ebergen, and M. I. Elmasry, "Modeling and comparing CMOS implementations of the C-element," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 6, no. 4, pp. 563–567, Dec. 1998.

[18] J. Sparso and J. Staunstrup, "Delay-insensitive multi-ring structures," *Integr. VLSI J.*, vol. 15, no. 3, pp. 313–340, Oct. 1993.

[19] C. D. Nielsen, "Evaluation of function blocks for asynchronous design," in *Proc. Conf. Euro. Design Autom.*, Sep. 1994, pp. 454–459.

[20] C. Jeong and S. M. Nowick, "Optimization of robust asynchronous circuits by local input completeness relaxation," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2007, pp. 622–627.

[21] C. Jeong and S. M. Nowick, "Block-level relaxation for timing-robust asynchronous circuits based on eager evaluation," in *Proc. 14th IEEE Int. Symp. Asyn. Circ. Syst.*, Apr. 2008, pp. 95–104.

[22] A. Bardsley, "Balsa: An asynchronous circuit synthesis system," M.S. thesis, Dept. Comp. Sci., Univ. Manchester, Manchester, U.K., 1998.
[23] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, "Balsa: A tutorial guide," M.S. thesis, Dept. Comp. Sci., Univ. Manchester, Manchester, U.K., 2006.
[24] C. D. Godsil and G. Royle, *Algebraic Graph Theory*. New York: Springer-Verlag, 2001.
[25] J. M. Robson, "Algorithms for maximum independent sets," *J. Algorithms*, vol. 7, no. 3, pp. 425–440, 1986.
[26] F. V. Fomin, F. Grandoni, and D. Kratsch, "A measure & conquer approach for the analysis of exact algorithms," *J. ACM*, vol. 56, no. 5, p. 25, 2009.
[27] T. H. Cormen, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2001.
[28] S. K. Bandapati, S. C. Smith, and M. Choi, "Design and characterization of convention self-timed multipliers," *IEEE Des. Test Comput.*, vol. 20, no. 6, pp. 26–36, Nov.–Dec. 2003.
[29] B. Hollosi, M. Barlow, F. Guoyuan, C. Lee, D. Jia, S. C. Smith, "Delay-insensitive asynchronous ALU for cryogenic temperature environments," in *Proc. 51st Midwest Symp. Circuits Syst.*, Aug. 2008, pp. 322–325.

Farhad A. Parsan (S'06) received the B.S. and M.S. degrees in electrical engineering from the Iran University of Science and Technology, Tehran, Iran, in 2005 and 2008, respectively, where he researched mixed-signal IC design. He is currently pursuing the Ph.D. degree with the University of Arkansas, Fayetteville.

He joined the Missouri University of Science and Technology, Rolla, in 2009, where he researched asynchronous IC design. He continued his research on asynchronous IC design by joining the University of Arkansas in 2010. His current research interests include asynchronous logic design, VLSI design and verification, design for testability, CAD tool development, and computer architecture.

Waleed K. Al-Assadi (SM'09) received the Ph.D. degree in electrical engineering from Colorado State University, Fort Collins, in 1996.

He is currently a Faculty Member with the Department of Electrical and Computer Engineering, University of South Alabama, Tuscaloosa. He has seven years of industrial experience with AMD and IBM Microelectronics. His current research interests include VLSI systems designs, design for test, security and trustworthiness of hardware, asynchronous paradigms, wireless sensors, reliability of nanotechnology systems, and embedded systems. He holds four U.S. patents.

Scott C. Smith (SM'06) received the B.S. degrees in electrical engineering and computer engineering and the M.S. degree in electrical engineering from the University of Missouri, Columbia, in 1996 and 1998, respectively, and the Ph.D. degree in computer engineering from the University of Central Florida, Orlando, in 2001.

He was an Assistant Professor with the University of Missouri, Rolla (now Missouri University of Science and Technology) in August 2001, was promoted to an Associate Professor in March 2007 (effective September 2007), and is currently an Associate Professor with the University of Arkansas, Fayetteville. He has authored 14 journal publications, 55 conference papers, and holds five U.S. patents, all of which can be viewed on his website: http://comp.uark.edu/ smithsco/. His current research interests include computer architecture, asynchronous logic design, CAD tool development, embedded system design, VLSI, FPGAs, trustable hardware, self-reconfigurable logic, and wireless sensor networks.

Dr. Smith is a member of Sigma Xi, Eta Kappa Nu, Tau Beta Pi, and the American Society for Engineering Education.