Dual-Rail Asynchronous Logic Multi-Level Implementation

Igor Lemberski*(corresponding author), Petr Fišer**

** Baltic International Academy, Lomonosova 1/24, Riga, LV-1019, Latvia, e-mail: Igor.Lemberski@bsa.edu.lv, ph.: +371 67 10 06 26, fax : +371 67 24 12 72*
*** Czech Technical University in Prague, FIT, Dept. of Digital Design, Prague, Czech Republic e-mail: fiserp@fit.cvut.cz*

Abstract. A synthesis flow oriented on producing the delay-insensitive dual-rail asynchronous logic is proposed. Within this flow, the existing synchronous logic synthesis tools are exploited to design technology independent single-rail synchronous Boolean network of complex (AND-OR) nodes. Next, the transformation into a dual-rail Boolean network is done. Each node is minimized under the formulated constraint to ensure hazard-free implementation. Then the technology dependent mapping procedure is applied. The MCNC and ISCAS benchmark sets are processed and the area overhead with respect to the synchronous implementation is evaluated. The implementations of the asynchronous logic obtained using the proposed (with AND-OR nodes) and the state-of-the-art (nodes are designed based on DIMS, direct logic, NCL) network structures are compared. A method, where nodes are designed as simple (NAND, NOR, etc.) gates is chosen for a detailed comparison. In our approach, the number of completion detection logic inputs is reduced significantly, since the number of nodes that should be supplied with the completion detection is less than in the case of the network structure that is based on simple gates. As a result, the improvement in sense of the total complexity and performance is obtained.

Keywords: asynchronous logic, decomposition, multi-level implementation, Boolean network, node.

1. Introduction

Asynchronous logic attracts an increasing interest of designers because asynchronous (delay-insensitive - DI) circuits are extremely *robust*. This means, the design is able to adapt to variations of manufacturing process parameters, gate and wire delays, temperature changes, noise, etc. [1]. The correct function is guaranteed, only the operational speed changes adaptively. Furthermore, a DI paradigm is very similar to synchronous one and generally, the DI design process follows the same steps as in synchronous logic design. As a result, it enables the developed DI design flow to be more easily incorporated into the design industry, since the tools and design processes are familiar to designers. The DI design process can be easier implemented, since a minimal delay analysis is required to ensure the circuit correct behavior. DI paradigm has additional advantages in designing complex circuits including substantially reduced crosstalk between analog and digital circuits, ease of multi-rate circuits cooperation, facilitation of component reuse.

The general disadvantages of DI asynchronous circuits with regard to the synchronous ones are high area and huge power consumption overheads, although the thermal distribution is uniform across the chip.

We propose a synthesis flow of multi-level DI dual-rail implementation. It is based on exploiting synchronous logic synthesis tools to produce a single-rail Boolean network of a two-level (AND-OR) nodes, and further transformation of the network into a dual-rail one. Based on results [18], each node is designed as a hazard-free structure. Finally, the technology-dependent mapping procedure is applied. For the comparison, several state-of-the-art methods are considered, where nodes are designed based on DIMS [13], direct logic [11], and NCL [15]. For the detailed comparison, the method [17], where each single-rail Boolean network node is designed as a simple gate (NAND, NOR, etc.) was chosen. We believe that this method is the closest one to our approach. Although [17] is supposed for designing some other class of circuits, it is clear that the method can easily be adapted for DI logic synthesis. Indeed, the Boolean network [17] is designed as a dual-rail hazard-free logic. The indication of the new input state and internal stability can be done using the completion detection (CD) logic proposed in this paper.

The main disadvantage of the method [17] is a large number of nodes that should be supplied with the CD – the completion detection must be provided for each simple gate. It is not the case of our approach, where the number of nodes of the synthesized Boolean network is significantly less than in [17]. Therefore, in our approach, the CD logic complexity is reduced, although the Boolean network implementation complexity may be slightly increased (to ensure hazard-free implementation of the AND-OR nodes). As a result, the improvement in sense of the total complexity and performance is obtained. The other approaches to CD

optimization can be found in the literature. Namely, in [30], the optimization method based on the evaluation of the gates relative timing was proposed. In [31], the method in a cost-aware manner was described.

The rest of the paper is organized as follows. In Section 2, the review of the related works is given. The information and notations regarding dual-rail logic is presented in Section 3. Also, details of DI logic behavior rules that are based on Seitz's strong and weak constraints are described. Section 4 is devoted to the description of the model with modified weak constraints. Next, the node minimization constraint to ensure hazard-free implementation is formulated and the structure of the dual-rail network is proposed. Examples illustrating the state-of-the-art and our approaches are given. It is shown that our approach produces networks with significantly less number of signals the CD logic is supplied with. Section 5 describes the technology-independent and technology-dependent synthesis procedure. Experimental results are given in Section 6. Statements summarizing the results conclude the paper.

2. Related work

The asynchronous logic is classified depending on the mode of interaction with the environment. In the *input-output mode*, the environment is allowed to change the input state once a new output state is produced. There is no assumption about internal signals and the environment is allowed to change the input state before the circuit is stabilized in response to the previous input state.

In the *fundamental mode* (assumed in this paper, too), the logic operates based on the following discipline: the environment changes the input state once the output state has changed in response to the current input state and each gate inside the circuit is stable. Both design methodologies assume either bounded (a maximal value is known) or unbounded (a maximal value is unknown) gate and wire delays.

In case of the fundamental mode with *bounded delays*, the moment when the environment may change the input state is estimated based on the worst case propagation delay [3]. Within this model, only one input signal can be changed at a time. In [4], a generalized fundamental mode was proposed, where multiple input changes are allowed during a narrow time interval. For such a mode, a method of hazard-free two-level implementation was published [5]. A multi-level hazard-not-increasing transformation is applied to optimize the implementation [6]. Methods of hazard-free technology mapping were proposed in [7] and [8].

In case of the *unbounded delays*, the asynchronous logic should be capable of:

1) recognizing the moment when a new input state (generated by the environment) appears on the inputs and the moment when the circuit generates a new output state in response to the input one;

2) notifying the environment of new input and output states. After receiving the notification, the environment can generate the next input state.

To solve this problem, m-of-n codes of length n are used for states encoding, where each valid state is represented by ones in m positions and zeroes in the rest of (n-m) ones [2]. Among them, 1-of-2 (or dual-rail) as well as 1-of-4 encodings have been of special interest. The other 1-of-n encodings are rather expensive, since their implementation requires more wires than the dual-rail one. In this paper, the dual-rail state encoding is used.

A four-phase behavior discipline is supposed: to change an input state, the environment should reset it first (change to so called *spacer state*). The output state resets too, as a result. After that the environment sets a new input state. It implies a new output state. The behavior rule is based on Seitz's strong or weak constraints [9], [10]. Under the strong constraints, each output changes its state only when all inputs have changed their state. Under the weak constraints, some outputs are permitted to change their state when some (not all) inputs have changed their state. In the case of strong constraints, output signals also serve as the completion detection ones and indicate the moment when both internal and output signals become stable. In case of weak constraints, output signals may also serve as the completion detection, if they are able to indicate the state of all input signals. Otherwise, an additional completion detection block is required to ensure a proper indication [11]. In [12], the distribution of the completion detection between the outputs is proposed to minimize the implementation cost. Also, the completion detection must indicate the moment when internal signals become stable.

The dual-rail implementation under the four-phase discipline is based on Delay-Insensitive Minterm Synthesis (DIMS) technique [13]. Within it, a function is implemented as a two-level structure with C-elements on the first level and an OR gate on the second one. The DIMS cost is very high, since the minimization of the number of product terms is not allowed. Therefore, 2^k minterms (where k is the number of C-element inputs) must be generated to implement each function's positive and negative forms, where each minterm is implemented using a k-input C-element. Finally, the C-element is more complex than a simple gate.

The implementation of the two-level C-OR logic as a single CMOS gate (Direct Logic) significantly reduces the area [11].

A similar approach is based on using threshold functions (Null Convention Logic-NCL). NCL circuits are designed based on 27 library gates that are capable of implementing any function of four or less inputs [15]. In the case of dual-rail logic, each literal is considered as a separate variable. Therefore, *not any* single-rail function of more than two inputs can be implemented in NCL; the feasibility of a function implementation depends on the number of literals

in its dual-rail representation.

Multi-level implementations of the dual-rail asynchronous logic were proposed in [14] and [17]. These methods are based on the initial circuit decomposition into simple (AND, OR, NOR, NAND, etc.) two-input gates. Further, each gate is mapped into DIMS [13] or implemented by a threshold gate [14]. It results in not only high complexity of the circuit implementation (in sense of the area), but also in a low performance, since each simple gate (single level structure) is implemented as a two-level (AND-OR, C-OR) structure. In [17], each simple gate is doubled to implement dual-rail logic. Compared to the synchronous implementation, the circuit cost doubles.

Desynchronization [16] is a modern paradigm that is based on adopting synchronicity to the asynchronous logic design. If bounded delays are supposed, matched delays are introduced for the synchronization purpose. In case of unbounded delays, extra completion detection logic should be present to indicate the circuit stability. A network of local controllers is designed to provide proper local synchronization signals, resulting in an additional area penalty. Finally, the circuit is equipped with output latches and a new output state is available only once a latch signal enables.

Our approach is based on the combination and extension of methods [15, 17]. Namely, we propose the Boolean network multi-level implementation, where complex gates of general nature (represented as a two-level structure) are produced and transformed into the dual-rail logic.

3. PRELIMINARIES

3.1. Single and Dual-Rail Encoding

Let $F = \{f_1, f_2, \ldots, f_q\}$ be a multi-output function of n primary inputs X: $X = \{x_1, x_2, \ldots, x_n\}$ and q primary outputs. Let $Y = \{y_1, y_2, \ldots, y_m\}$, $f_1, f_2, \ldots, f_q \in Y$, $m \geq q$, be a set of single-output Boolean nodes obtained as a result of a decomposition of F. Each node function y_c depends on given k or less number on inputs: $y_c = y_c(z_{c1}, z_{c2}, \ldots, z_{ck})$, $|y_c| \leq k$, $z_{c1}, z_{c2}, \ldots, z_{ck} \in \{X \cup Y \setminus \{f_1, f_2, \ldots, f_q, y_c\}\}$. We call it a single-rail multi-level implementation. In [17], the node is a simple gate (NOR, NAND, etc). In our case it may also be a two-level (AND-OR) complex node (Figure 1a). Usually, a single-rail representation is obtained as a result of the synchronous logic synthesis and optimization.

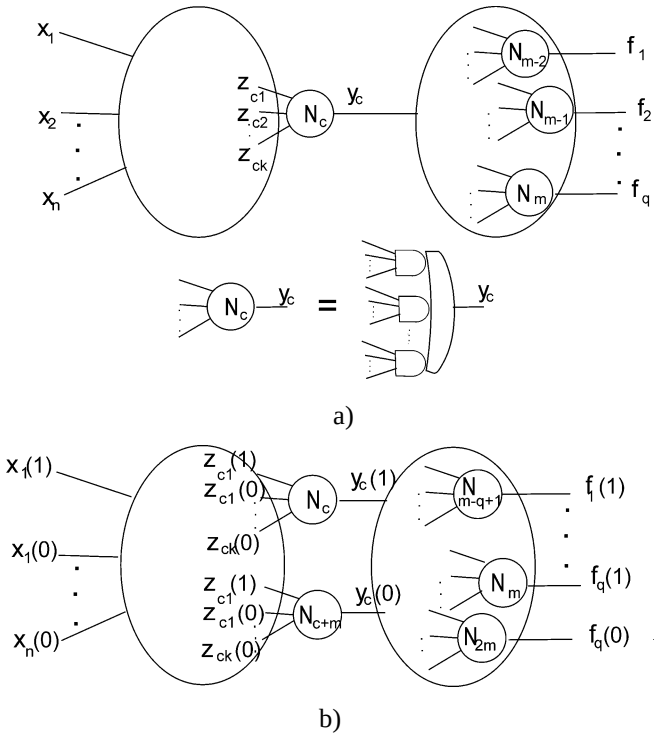

Figure 1. Single– (a) and dual-rail (b) multi-level Boolean network

In dual-rail logic, it is supposed that each node input from the set $Z = \{z_{c1}, z_{c2,\ldots}, z_{ck}\}$ and the node output y_c may be in one of these three states: states 1, 0 (so called working states) or undefined (spacer state). To implement a three-state input z_i, two signals $z_i(1)$ and $z_i(0)$ are introduced, where $z_i(1) = 1$ and $z_i(0) = 0$, if z_i is in state 1, $z_i(1) = 0$ and $z_i(0) = 1$ if z_i is in state 0, $z_i(1) = z_i(0) = 0$ if x_i is in the spacer state. The combination $z_i(1) = z_i(0) = 1$ is not allowed. Similarly, to implement a three-state node function, the function y_c, $c = 1, 2, \ldots, m$, should be represented in both positive $y_c(1)$ and negative $y_c(0)$ forms. If $y_c(1) = 1$, $y_c(0) = 0$, then the function y_c is in state 1, if $y_c(1) = 0$, $y_c(0) = 1$, then the function y_c is in state 0, if $y_c(1) = y_c(0) = 0$, then the function y_c is in the spacer state. The combination $y_c(1) = y_c(0) = 1$ is not allowed. To change the input state, the environment should reset it first to the spacer state and after that set it to the proper working state. In the reset phase, the output state changes from the working state to the spacer one and in the set phase the new output state is recognized.

In the dual-rail logic (Figure 1b), each function y_c is represented as a pair: $y_c = (y_c(1), y_c(0))$, where $y_c(1)$, $y_c(0)$ describe its ON-, OFF- sets ($y_c(0)$ can be generated as a complement to the ON-set). Therefore, each function y_c can be represented as a Sum-Of-Product terms (SOP) of both positive $y_c(1)$ and negative $y_c(0)$ forms: $y_c(1) = t_1 + t_2 + \ldots + t_s$, $y_c(0) = t_{s+1} + t_{s+2} + \ldots + t_p$, $c = 1, 2, \ldots, m$, $p \leq 2^k$, where t_i are product terms containing n or less literals, $i = 1, 2, \ldots, p$, $t_i \cap t_j = \emptyset$, for $\forall(t_i, t_j)$: $t_i \in y_c(1)$ and $t_j \in y_c(0)$.

3.2. Strong and weak constraints

In the fundamental mode, the DI logic operates under so called *strong* or *weak constraints* [10]. Timing diagrams depicting behavior rules under strong and weak constraints are presented in Figure 2.

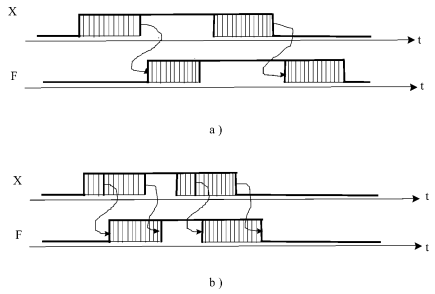

Figure 2. Behavior rules under strong (a) and weak (b) constraints

The diagrams depict inputs X and outputs F state changes in time t along with input/output states dependency. Each input and output may be either in a working (high level) or a spacer (low level) state. Vertical shading depicts time intervals, where input and output states are going from the spacer to a working state and vice versa.

Under the strong constraints (Figure 2a), the behavior rule is as follows:

1. If all inputs are in the spacer state, then all outputs are going to the spacer state;
2. If all inputs are in a working state, then all outputs are going to a working state.

Under the weak constraints (Figure 2b), some outputs are permitted to change their states when some (not all) inputs have changed their states:

1. If some inputs are in a working state, then some outputs are going to a working state;
2. If all inputs are in a working state, then all outputs are going to a working state;
3. If some inputs are in the spacer state, then some outputs are going to the spacer state;
4. If all inputs are in the spacer state, then all outputs are going to the spacer state.

In both cases, it is supposed that the states of *all outputs* depend on the state of *all inputs*. However, in some cases, the state of outputs can be determined based on the state of some (not all) inputs. For example, consider a function $f(1) = x_1(1) + x_1(0)x_2(0)$. If $x_1(1) = 1$, then independently of the input x_2 state, the function $f = 1$. Based on this observation, one can modify the (weak) constraints as follows: if some inputs are in the working/spacer state then some *or even all outputs* are going to the working/spacer state. In Section 4 it will be shown, that under such a modification, the product term minimization is allowed.

4. MODEL BASED ON MODIFIED WEAK CONSTRAINTS

4.1. Behaviour Rule

To design the minimized two-level AND-OR logic, we introduce the model with modified weak constraints [18] under the following behavior rules (Figure 3):

1. If some inputs are in a working state then all outputs are going to a working state;
2. If all inputs are in a working state then all outputs remain in a working state;
3. If some inputs are in the spacer state then all outputs are going to the spacer state;
4. If all inputs are in the spacer state then all outputs remain in the spacer state.

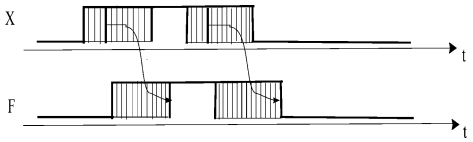

Figure 3. Behavior rule under modified weak constraints

The structure consists of two blocks (Figure 4): a two-level AND-OR structure and the CD logic. Since minimization is allowed, the AND-OR structure is implemented by less than 2^n product terms ($p < 2^n$ - Figure 4) and product terms may contain less than n literals: $|S(t_k)| \le n$, where $S(t_k)$ is a set of term t_k literals (input signals), $k = 1, 2, \ldots, p$. Note, that outputs are not capable of indicating the states of all inputs, because output states may depend on some (not all) inputs. Furthermore, since multi-output functions are supposed, an additional signal is required to indicate the moment when all outputs are in a proper state (working or spacer). Therefore, the completion detection logic with an output signal D is introduced. The signal D switches on, when both inputs and outputs are in a working state.

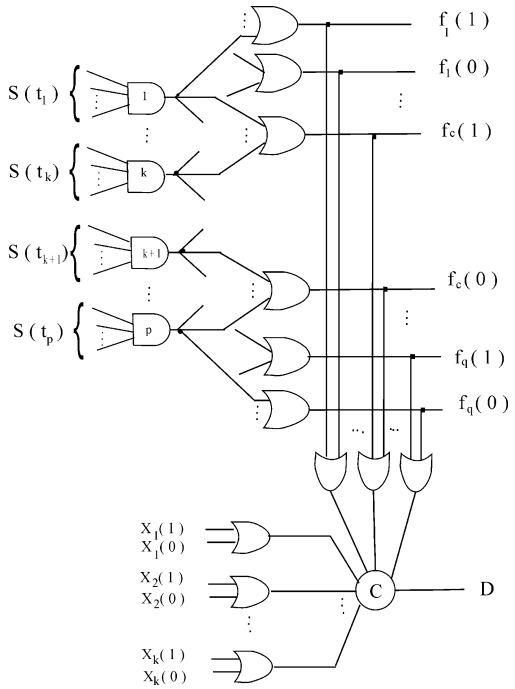

Figure 4. Dual-rail two-level logic

4.2. Monotonicity and Hazard-Free Implementation

The proposed structure is based on the concept of *monotonicity* of the nodes introduced in [17] and a *hazard-free* implementation of each two-level (AND-OR) node proposed in [18].

4.2.1 Monotonicity

A node implementing the function $y_c = y_c(z_{c1}, z_{c2}, ..., z_{ck})$ is *positive*, if for each input z_c , $z_c \in \{z_{c1}, z_{c2}, ..., z_{ck}\}$ in its local fan-in it holds the following: if the input z_c is set to 1 (0), then the function y_c is set to 1 (0). A node implementing function y_c is *negative*, if for each input z_c in its local fan-in it holds the following: if the input z_c is set to 1 (0) then the function y_c is set to 0 (1). The node is *monotonic* if it is either *positive* or *negative*.

The node monotonicity is easily achieved by the dual-rail encoding [17].

4.2.2 Hazard-Free Implementation

To satisfy the proper behavior rules (see Subsection 4.1) and therefore ensure the hazard-free implementation, product terms within both ON- and OFF-sets must be orthogonal, i.e., a Disjoint Sum-Of-Products (DSOP) is supposed for their implementation [18], [29]. Based on [18], the following theorem is valid:

Theorem 1. The behavior of any function f_c (Figure 4), $f_c \in F$, does not violate the behavior rule (Section 4.1) iff: $t_i \cap t_j = \emptyset$ (the terms are *orthogonal*), for $\forall(t_i, t_j)$: $t_i, t_j \in f_c(1)$ and $t_i, t_j \in f_c(0)$.

Proof. Necessity. Suppose, that $t_i \cap t_j \neq \emptyset$. Terms t_i, t_j can't belong to different sets $f_c(1)$, $f_c(0)$, since $f_c(1) \cap f_c(0) = \emptyset$. Therefore, either $t_i, t_j \in f_c(1)$ or $t_i, t_j \in f_c(0)$. First, suppose $t_i, t_j \in f_c(1)$. Consider the circuit fragment containing AND_j gates connected to an OR gate implementing the function $f_c(1)$ (Figure 5a). Let $S(t_i)$ be a set of term t_i literals (input signals). Define a joint set $S(t_{ij})$: $S(t_{ij}) = S(t_i) \cup S(t_j)$. Let S(X) be a set of input signals that switch on once inputs switch to a given working state. For example, given input working state is: $x_1(1) = x_2(0) = 1$. Then $S(X) = \{x_1(1), x_2(0)\}$. Suppose: $S(t_{ij}) \subseteq S(X)$ and: 1) a signal $x_l(u)$ has the longest switching delay among the signals of set $S(t_i)$; 2) a signal x_r has the longest switching delay among the signals of set $S(t_j)$, $x_l(u) \in S(t_i)$, $x_r(u) \in S(t_j)$, $u \in \{0,1\}$. When the signal $x_l(u)$ switches on, then the AND_j gate output switches on. Similarly, when the signal $x_r(u)$ switches on, then the AND_j gate output switches on. As a result, signal 1 propagates through two paths: AND_i - OR, AND_j - OR. Suppose that the sum of the signal $x_l(u)$ switching delay and the path AND_i-OR delay is shorter than the sum of the signal $x_r(u)$ switching delay and the path AND_j - OR delay. In this case, the signal 1 propagating through the path AND_i - OR switches the output $f_c(1)$ on (the output f_c goes to the working state).

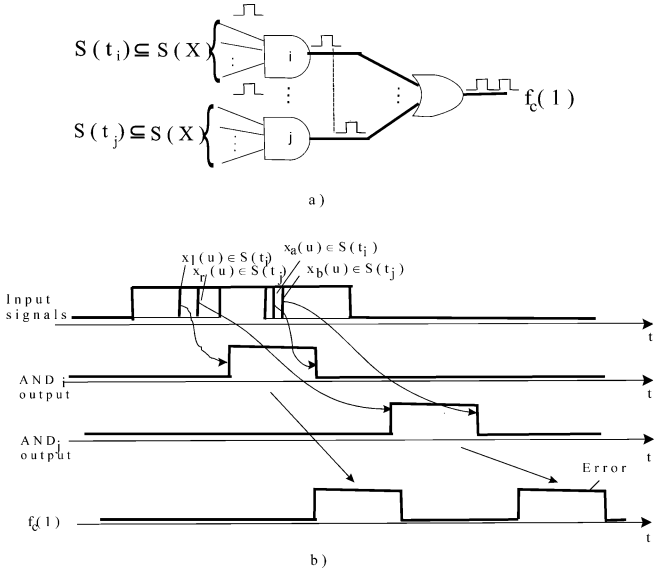

Figure 5. Necessity conditions: a) circuit, b) timing diagram

Now suppose that any signal $x_a(u) \in S(t_i)$ switches off (the input x_a goes to the spacer state). It implies switching the AND_i gate output off. As a result, the signal 0 propagates through the path AND_i-OR and the function $f_c(1)$ switches off (the output f_c goes to the spacer state). Due to a longer propagation delay, the signal 1 propagating through the path AND_j - OR may switch the output $f_c(1)$ on again and later off due to switching any signal $x_b(u) \in S(t_j)$ off (erroneous pulse - Figure 5b). It violates the behavior rule ("if some inputs are in the spacer state then some outputs are going to the spacer state and remain in this state"). The above conclusion is valid, if $t_i, t_j \in f_c(0)$.

*Sufficiency***.** Again consider the same circuit fragment

(Figure 6a). Suppose: $t_i \cap t_j = \emptyset$, t_i, $t_j \in f_c(1)$. Then, $S(t_j) \not\subset S(X)$ because of $t_i \cap t_j = \emptyset$ and in contrast to the previous case, there is only one path (Figure 6a, bold line) for the signal propagation to the output.

Therefore, if signals from the set $S(t_i)$ switch on, then the AND_i gate output switches on and, in turn, the output $f_c(1)$ switches on (Subsection 4.1, rule 1). Once any signal $x_a(u) \in S(t_i)$ switches off then the output $f_c(1)$ switches off (Subsection 4.1, rule 3) (Figure 6b). As a result, the behavior of any function $f_c \in F$ doesn't violate the behavior rule (Subsection 4.1). This conclusion is valid for a function $f_c(0)$, if $t_i, t_j \in f_c(0)$. ■

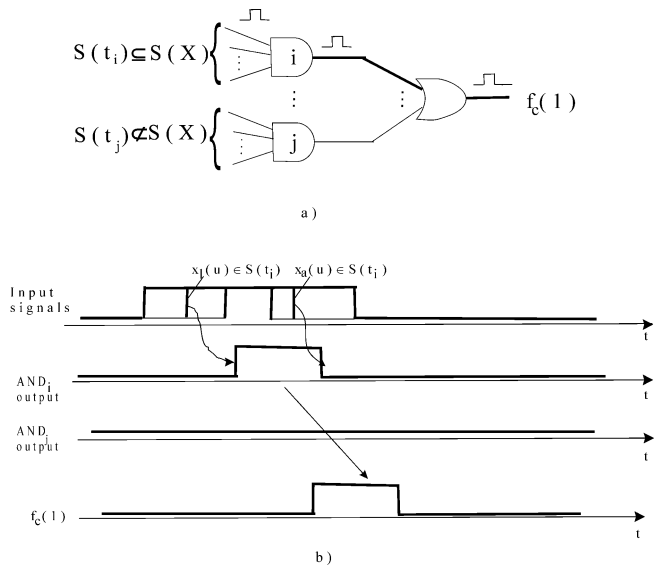

Figure 6. Sufficiency conditions: a) circuit, b) timing diagram

4.3. Dual-Rail Multi-Level Network

Given a multi-level Boolean network (Figure 1a) obtained as a result of synchronous synthesis. The network is transformed into a dual-rail (Figure 1b) hazard-free one (the procedure will be given in the Subsection 5.1). The structure consists of two blocks (Figure 7): the functional one implementing the Boolean network as a multi-level logic using two-level AND-OR single-output nodes having a fan-in limited to *2k* (remember, once given a single-rail node, then in dual-rail each input is represented as two separate inputs) and the CD logic that is obtained by merging the CDs of all nodes. The CD should indicate the proper state (working or spacer) of the network and of primary inputs and outputs. The CD logic is based on *(n+m)* C-elements together with *(n+m)* two-input OR gates, where *n* is number of primary inputs and *m* the number of nodes (including the ones generating *q* primary outputs). The CD signal *D* (Figure 7) is going up, when both primary inputs and node outputs are all in a working state and going down when all the signals mentioned are in the spacer state.

Note, that this structure guarantees DI implementation even if the functional block is designed as a network of simple nodes as proposed in [17]. It gives an opportunity to consider the implementation (Figure 7) exploiting dual-rail network from [17] as a state-of-the-art and compare the proposed DI implementation with it. As mentioned before, our approach benefits from the significantly less number of CD input signals. It results in reduction of the complexity of the CD block. Although in the case of two-level nodes the functional block complexity is slightly increased, the total complexity is reduced.

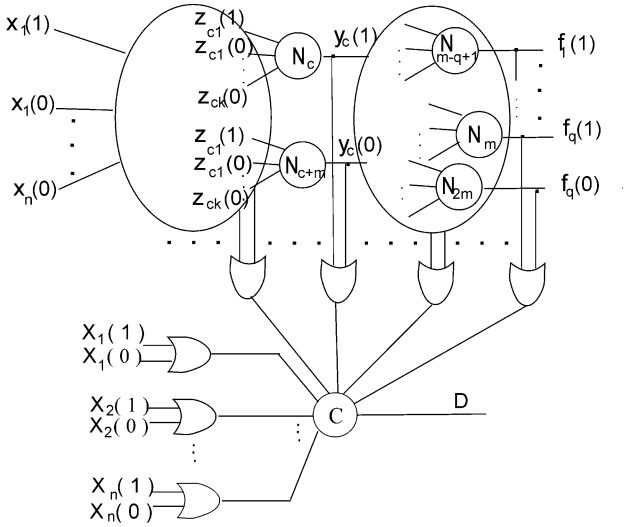

Figure 7. Dual-rail multi-level Boolean network with the CD logic

4.4. Examples

Let us consider an example of two implementations. The first implementation is based on the method [17], where the functional logic is represented as a Boolean network of simple (NAND, NOR, etc.) gates. The second one is based on the approach that we propose, where nodes may be implemented as a two-level logic that satisfies the constraint formulated in Theorem 1.

Given a single-rail circuit (Figure 8a), where c' means a complement of the signal c. Following [17], the dual-rail logic is produced and the CD logic is designed as given in Subsection 4.3. It results in the implementation shown in Figure 9. One can immediately see that the number of CD inputs is 8. However, one can easily notice that a group of NAND gates can, in fact, be treated as a three-input complex NAND-NAND node (shown in Figure 8b in bold), which is equivalent to an AND-OR structure. Such a node can be implemented as a four-input complex node (Figure 10) in dual-rail logic (remember, an input and its complement are treated as two separate inputs). Note, that it is described by the function with orthogonal terms (ce + c'd) and

therefore is hazard-free (Theorem 1). The complement node (ce' + c'd') is also described by a function with orthogonal terms. Note, that the complement function obtained using the transformation, proposed in [17] and shown in Figure 9, is as follows: ((e' + c')'(c + d')')' = ce' + e'd' + c'd' and contains an additional term e'd'. However, these two functions are equivalent, because in the working state the term e'd' is redundant, but in the spacer state it produces a zero value, because all signals are in the spacer state.

One can see that the functional block complexity remains the same. However, the CD block implementation will be simpler, since it is supplied with 6 inputs only.

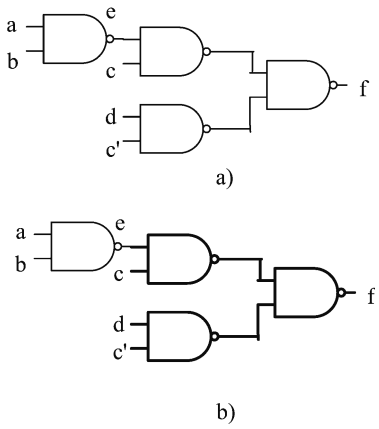

Figure 8. The circuit representation: a) as a simple gate logic; b) with a two-level complex node

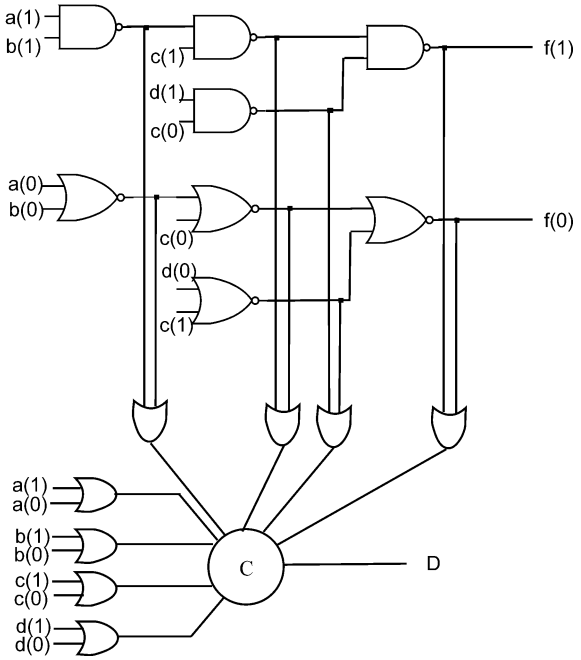

Figure 9. The implementation based on simple gates

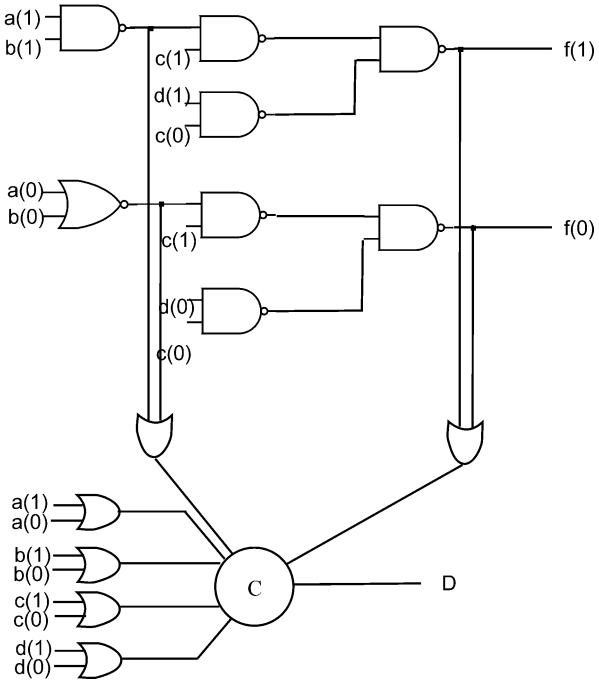

Figure 10. The implementation based on complex gates

5. SYNTHESIS

5.1. Technology-Independent Synthesis

The proposed procedure of synthesis of the multi-level dual-rail logic with AND-OR nodes is based on tools ABC [19] used for multi-level synthesis, Espresso [20] minimizing two-level (represented as AND-OR structure) nodes, and DSOP [21] used to obtain two-level nodes with orthogonal terms.

We start with an initial circuit description. First, an ABC script is applied to it, to obtain a multi-level single-rail Boolean network with the fan-in of each node limited to a given *k*. For this purpose, we have decided to use a LUT mapping synthesis process, since each LUT is actually represented as a single-output AND-OR node with a limited number of inputs (in the BLIF format generated by ABC [22]).

We have used a sequence of ABC commands recommended for the LUT synthesis in the ABC reference guide [19] (Figure 11). The "`choice`" script performs a technology-independent optimization of the network. Then, the "`fpga`" command maps the network into *k*-input LUTs. Finally, possibly redundant LUTs are removed by "`lutpack`". This command sequence was repeated 10-times, to obtain better results.

```
choice
fpga -K k
lutpack
```

Figure 11. The LUT mapping script. Substitute *k* for the maximum node fan-in

An alternative way of producing a network of arbitrary *k*-input nodes is to construct a technology library of *all* *k*-input functions and perform the technology (standard-cell) mapping by the ABC command "`map`". Then the mapped library gates can be transformed back into their SOP representation by, e.g., the "`sweep`" command. The overall synthesis process will be then as follows:

```
read_library k-gates.genlib
choice
map
sweep
```

Figure 12. The standard cells mapping script. Substitute *k* for the maximum node fan-in

Apparently, such an approach is feasible for smaller *k's* only (up to 4), due to a double-exponential growth of the number of functions. However, using this approach at least for $k = 2$ was a necessity, since the "`fpga`" command does not support 2-LUT mapping.

After the single-rail synthesis, transformation into a dual-rail network is done. The procedure includes the following steps:

1) a complement (OFF-set) node for each (ON-set) node is computed by the algorithm used in Espresso [20]. As a result, the number of nodes is doubled;

2) each ON-set and OFF-set node is minimized by Espresso, since the LUT mapping process does not care about the SOP minimization and the Espresso complementation algorithm does not always return minimum results;

3) each AND-OR node is orthogonalized by DSOP [21], to ensure the hazard-free behavior (Theorem 1). Simple nodes (like AND, OR, NAND, XOR etc) need not be orthogonalized, since they are implemented as single gates;

4) single-rail signals are replaced by dual-rail ones. As a result, each node function may depend on up to *2k* inputs (since each signal and its complement one are represented as separate rails).

Another issue emerging in the dual-rail logic design with two-level nodes is a possibility of sharing of product terms between the nodes (except of pairs of ON- and OFF-set nodes, where the OFF-set node is a complement to the ON-set node, since such two sets must be orthogonal). In the dual-rail network, shared product terms are identified and implemented only once.

5.2. Technology-Dependent Synthesis

In the case of technology-dependent synthesis, we estimate the complexity of the functional network and the completion detection logic separately. Then, the total complexity is calculated. To avoid additional inverters and therefore decrease the implementation complexity, we use negative (NAND-NAND) gates instead of AND-OR ones in the functional block and NOR gates instead of OR ones in the completion detection logic. As a result, the signal D = 1, when all inputs and outputs are in the spacer state and D = 0 for the working state.

We suppose a library of *k*-input NAND and NOR gates, *k*-input C-elements, 2-input XOR gates, and inverters. The network obtained as a result of the technology-independent synthesis (see Subsection 5.1) may include logic that can't be mapped into the above library, namely the number of the node second level NAND gate inputs may exceed the given *k*, since the node may be described by more than *k* terms (up to $2^k/2$, for the XOR function). In this case the NAND gate is decomposed into a tree of NAND gates of *k* or less inputs and inverters are placed to ensure the functionality. Note, that the node logic remains hazard-free, since the transformation doesn't violate the orthogonality of the product terms. Therefore, once the NAND gate is decomposed and mapped into a tree of NAND gates, the output of the last NAND gate

only should be connected to completion detection logic. The technology-dependent implementation (with the technology limit $k = 2$) of a node having a 3-input NAND gate is shown in Figure 13.

Next, the number of C-element inputs is $(n+m)$, which will definitely exceed k for non-trivial designs. Thus, the $(n+m)$ input C-element is transformed into a delay-optimum tree structure of k–input C-elements (Figure 13) by a topological traversal of the dual-rail network. The topological ordering is determined by computing the actual signal arrival times of each complex node output. Outputs of nodes with the lowest arrival times and primary inputs, which have zero arrival time by definition, are connected to C-elements first. The topological ordering is updated by including the signal arrival times of the produced C-elements outputs as well as removing connected node outputs and primary inputs. The procedure is repeated, until primary output D is constructed.

The idea behind this procedure is as follows: the higher input signal arrival time is, the closer to the C-elements structure top gate it should be connected, and vice versa. This decreases the signal propagation time. The best possible arrival time of D is obtained this way. Note, that although the CD logic is a tree, it may not be a balanced tree. The signals arrival times affect significantly the tree structure.

Finally, note that NAND gate decomposition into a tree of k - (or less) input NANDs and inverters is based on the same iterative procedure as for C-elements.

5.3. State-of-the-Art Synthesis for Comparison

For the sake of a just comparison, we have implemented the approach described in [17] as follows.

First, we have processed the initial network by the ABC "`choice`" script and mapped onto a library of standard k-input gates (AND, OR, NAND, NOR, XOR, XNOR) by the "`map`" command. Only 2-input XOR and XNOR gates were used, since standard cell libraries usually do not contain XORs of more inputs. The "`choice; map`" sequence was repeated 10-times, as in the proposed method.

The dual-rail logic was obtained by duplicating the network, while substituting all the gates by their negated counterparts.

Technology-dependent synthesis is guaranteed, no gate having more than k inputs may emerge, except of the C-elements structure, which is implemented in the same way as described in the Subsection 5.2.

Also, the network was implemented based on DIMS, direct logic [13] and NCL [15] for comparison. As for DIMS and direct logic, the LUT mapping procedure described in Subsection 5.1 was used, to obtain a network of k-input nodes. Since the NCL logic allows implementation of gates up to 4 inputs, only the 2-input nodes mapping process (see Subsection 5.1) was used. Generally, a k-input node, $k > 2$, may depend on $2k > 4$ variables in the dual-rail logic, thus cannot be implemented in NCL.

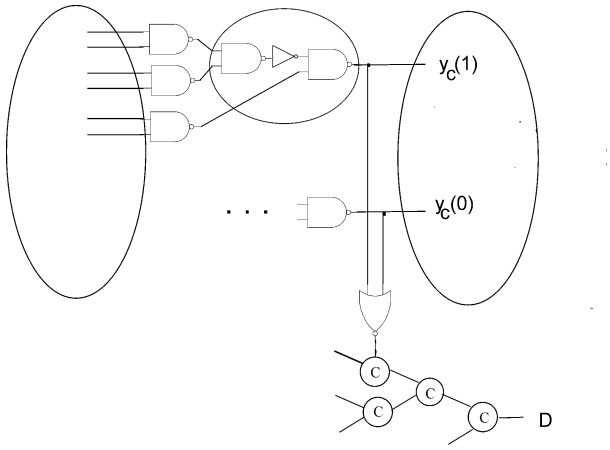

Figure 13. Technology-dependent implementation ($k = 2$)

6. EXPERIMENTAL RESULTS

6.1. Experimental background

We have processed the MCNC [23] and ISCAS [24], [25] sets of benchmarks, 228 circuits altogether. We evaluate the complexity (expressed as the number of the transistors) and the performance (by summarizing gates logical efforts [26] within the critical path) of the proposed asynchronous implementation of these circuits and compare it to the state-of-the-art [13], [15], [17].

We suppose a technology dependent synthesis (fan-ins of NAND gates and C-elements do not exceed a given k).

Suppose, given a library of k-input NAND and NOR gates, 2-input XOR gates and inverters. The gate complexity is estimated as follows: a k-input NAND or NOR gate requires $2k$ transistors, the inverter cost is 2 transistors and the 2-input XOR gate cost is 10 transistors, finally, the k-input C-element cost is $4(k+1)$ transistors [26].

Logical efforts of several typical gates are as follows: inverter: 1 (by definition, the logical efforts of other gates are calculated in the inverter logical effort units), NAND: $(k+2)/3$ units. NOR: $(2k+1)/3$ units, C-element: k units, where k is the number of gate inputs.

Consider a two-level complex node obtained as a result of technology-independent synthesis. Its k'-input OR gate (second level) is decomposed (by applying technology-dependent decomposition procedure - see Subsection 5.2) and mapped into a tree of NAND gates,

$k' > k$, where k is the technology limit. Within each iteration of this procedure, k inputs are removed and a single output is added and treated as a new input in the next iteration. Therefore, (k - 1) inputs are removed from the list until a list containing a single C-element output (primary output D) is constructed. As a result, the number of NAND library (k-input of less) gates required to map k'-input NAND gate is calculated using following formula:

$$\left\lceil \frac{k'-1}{k-1} \right\rceil \quad (1)$$

The number of inverters in the tree (between each pair of NAND gates) is:

$$\left\lceil \frac{k'-1}{k-1} \right\rceil - 1 \quad (2)$$

As a result, the total complexity of the k'-input NAND gate tree is:

$$2 \cdot \left((k+1) \cdot \left\lceil \frac{k'-1}{k-1} \right\rceil - 1 \right) \quad (3)$$

Now we calculate the complexity of the completion detection logic. To implement an (n+m)-input C-element, $4(n+m+1)$ transistors are required. If $n+m > k$, the C-element should be decomposed into a tree of k-input C-elements. In case of a balanced tree, its complexity is:

$$4 \cdot (k+1) \cdot \left\lceil \frac{m+n-1}{k-1} \right\rceil \quad (4)$$

To implement $(n+m)$ two-input NOR gates, $4(n+m)$ transistors are required. The total complexity of the completion detection logic for an n-input multi-level logic with m nodes is:

$$4 \cdot \left((k+1) \cdot \left\lceil \frac{m+n-1}{k-1} \right\rceil + m + n \right) \quad (5)$$

Note, that the complexity of the sequential logic memory (flip-flops, latches) is not included in the results; only combinational parts of the circuits are assumed.

The area of nodes implemented as DIMS and direct logic [13] was computed as follows:

TABLE I. COMPLEXITY OF OTHER IMPLEMENTATIONS

Inputs	DIMS [13]	Direct Logic [15]
2	24	22
3	64	34
4	160	54
5	384	90
6	896	158

The NCL implementation is based on a network containing NAND and XOR logic. Library gates TH22 and THand0 [15] are used to implement NAND logic and its complement. Gates TH22 and THand0 total complexity is 21 transistors. XOR logic and its complement implementation require two TH24comp gates with the total complexity 24 transistors.

6.2. Selection of k

The first issue addressed in the experiments is a proper choice of k (maximum node fan-in). Nodes with a high fan-in are difficult to be implemented in technology. On the other hand, small k's induce more nodes, which makes the completion detection logic more complex. Therefore, some kind of trade-off should be found.

We have synthesized the asynchronous dual-rail logic for all the 228 benchmark circuits, for k = 2, 3, 4, 5, 6. Both LUT and standard cells mapping processes were tested, for sake of a just comparison.

Note, that both processes do the same job: they construct a network of arbitrary k-input single-output functions. Just the mapping procedure is different.

The results are shown in TABLE II. Summary numbers of transistors and logical efforts for the 228 circuits are shown, for different k's and the two mapping processes ("k-map" and "k-fpga").

TABLE II. INFLUENCE OF K ON THE SIZE OF THE RESULTING LOGIC

Process	*Transistors*	*Logical effort*
2-map	3,191,136	6,586.33
3-map	2,361,086	5,343.67
4-map	**2,275,860**	4,942.67
3-fpga	2,271,142	5,376.33
4-fpga	**2,191,972**	4,955.33
5-fpga	2,349,356	4,952.00
6-fpga	2,615,018	4,933.33

We can see that using 4-input gates yield the smallest

summary area, both for standard cells and LUT mapping. We have found by a detailed analysis, that for 78% of circuits $k = 4$ gave better results than $k = 3$ and for 56% of circuits $k = 4$ was better than $k = 5$. Therefore, we can conclude that $k = 4$ should be considered for synthesis. However, $k = 3$ will be considered in the final experiments too, since the total area and performance differences from $k = 4$ are not so big.

Since the LUT mapping using the "`fpga`" command gave slightly better results than "`map`" (both in the area and performance), LUT mapping by "`fpga`" will be used in further experiments.

Obviously, the higher k, the less logical effort is. The figures are given in the "*Logical effort*" column.

6.3. Summary Results

Summary comparison results for all the 228 circuits are shown in TABLE III. and TABLE IV. for the area (in sense of the number of the transistors) and performance (in sense of logical effort), respectively. The total areas over all the 228 circuits for the proposed complex nodes, simple nodes, DIMS, direct logic, and NCL are shown in TABLE III. , from the second till the last column. Average size differences between the first method and the others are shown, too.

Similarly, the total logical efforts for the method [17] and the proposed one are given in TABLE IV. second and third columns. The average difference is shown in the last column. Positive difference values indicate an advantage of our method. Compared to the mentioned state-of-the-art methods, in many cases, our method gives better result in sense of the area.

We can see that the DIMS implementation is heavily inferior to all the others for $k > 2$, which is expectable.

The average size difference between our method and the direct logic is negative even for $k = 4$, however, we have observed that this is indifferently caused by several XOR-intensive circuits, whose direct logic implementation is much smaller. The absolute value of the number of transistors is less for our method.

Unfortunately it is impossible to make a representative comparison with NCL, since more complex NCL gates are not available.

Next, more detailed comparison w.r.t. the method [17], as the closest one to our approach will be done.

We can observe opposite tendencies in the area growth for [17] and our method: while the area of the method [17] decreases with increasing k, in our method it grows from $k = 3$. This can be explained by a simple fact: since only simple gates are used in the former one, the complexity of the asynchronous logic is proportional to the synchronous one. Designs with more-input gates are definitely simpler. On the other hand, complex nodes are used in our method. The number of SOP terms of each node may grow exponentially with k, similarly to DIMS. Therefore, the total complexity increases with k.

The average area consumed by our method is less than that of [17] for $k = 3$ and 4. The area differences obtained from all the 228 circuits are shown in Figure 14 for $k = 3$, where the area difference is most apparent. Each column in the figure represents one circuit.

The area was reduced in 76% of circuits, sometimes significantly (up to 84% for the *cordic* circuit, see TABLE V. We have observed that the area reduction mostly occurs for XOR-intensive and hard-to-synthesize circuits. The maximum area increase was only 25%.

Similar conclusion can be done regarding the logical effort. Although in 34% of circuits the logical effort is increased (by up to 28%), in most cases our method gives better results (see TABLE IV.). The highest average logical effort reduction (3.8%) is achieved for $k = 3$ as well. The logical effort reductions for all the 228 circuits are shown in Figure 15.

6.4. Detailed Results

Detailed results for some of the 228 benchmarks are shown in Tables V-VIII. We have selected 10 largest benchmarks, plus 8 benchmarks, where the highest area improvement is achieved. These two sets are separated by a double-line in the Tables.

Area calculation. TABLE V. Presents the area results for $k = 3$. After the benchmark name the numbers of its inputs (n) and outputs (q) are indicated. The complexities of synchronous implementations are shown in the next column, in terms of the number of transistors.

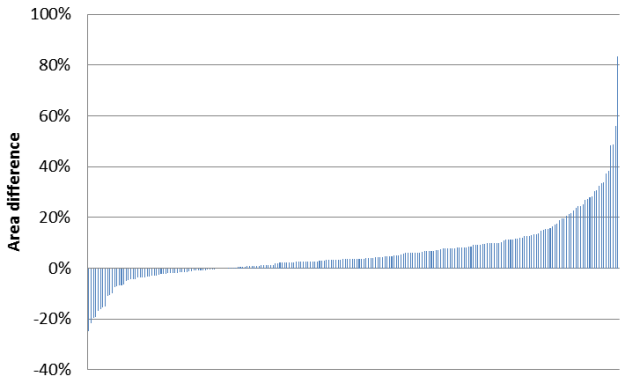

Figure 14. Area differences for 228 circuits, $k = 3$

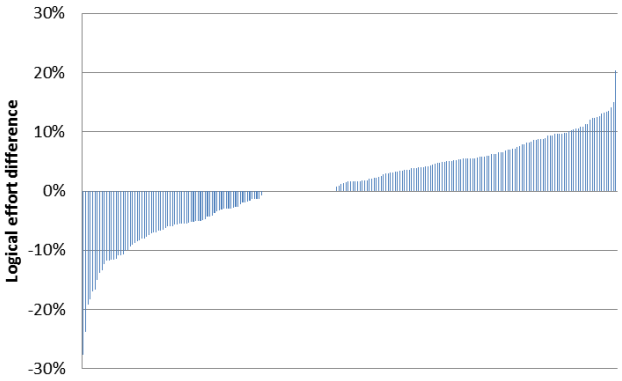

Figure 15. Logical effort differences for 228 circuits, $k = 3$

The complexity of the proposed asynchronous multi-level implementation of the benchmarks is shown next. Complexities of the functional logic ("*Funct. trans.*") and the completion detection logic ("*CD trans.*") are shown first, then the values are summed together to obtain the total asynchronous logic complexity ("*Total trans.*"). The area increase of the asynchronous logic w.r.t. the synchronous implementation is shown in the next column ("*Overh.*").

Complexities of the asynchronous multi-level implementation proposed in [17] are shown in the next triplet of columns. Again, the functional, completion detection and total complexities are given. The area reduction obtained by our method, w.r.t. [17], is shown in the last table column ("*Diff.*").

Average values of the area overhead and area difference computed from all the 228 circuits are shown in the last row.

Performance calculation. As already mentioned before, the logical effort calculation is done within a critical path by summarizing the logical efforts of gates on the path. The calculation results for $k = 3$ are shown in TABLE VI. The format is partially retained from TABLE V. , only the numbers of transistors are substituted by the logical effort values. Note that the CD logic performance is given as the number of levels. One can easily express it as the logical effort units multiplying the number of levels by k, since a single level (implemented by a k-input C-element) logical effort equals to k.

Note that the signals within the functional and CD logics propagate in parallel. The total logical effort is obviously higher than the functional logic one, however, generally, less than a sum of the logical efforts of the functional and CD logics.

Area and logical effort results for $k = 4$ are shown in TABLE VII. and VIII respectively.

7. Discussion and Conclusions

A novel synthesis flow of the dual-rail asynchronous multi-level logic is proposed. The logic is implemented as a monotonous multi-level network of minimized AND-OR nodes together with the CD logic. Each node is a hazard-free structure. We have formulated an additional minimization constraint (the SOP terms must be mutually orthogonal) for that purpose.

The proposed method offers a possibility of designing asynchronous circuits using both synchronous design tools and standard target technology used in synchronous designs. This is not the case of, e.g., NCL or direct logic, where special gates are used. The advantage of the proposed method over [17] is a reduction of the complexity of the CD logic. This can decrease both the total area and, more importantly, increase the performance, since the CD logic together with the functional logic form the critical path.

Additionally, since nodes are implemented as SOPs, sharing of terms as well as gate trees in the technology dependent implementation is possible. This is not the case of the other methods (NCL, direct logic), as each node is implemented as a monolithic gate there.

The MCNC and ISCAS benchmarks were processed and the complexity and performance of the logic obtained using the proposed and the state-of-the-art methods are compared.

We have found experimentally, that compared to [17], our method gives better results in sense of performance (improvement up to 4% for 4-input gates). Also, mappings into 3- or 4-input gates are the most efficient ones, in sense of the area (compared to [17], the improvement is more than 6% for 4-input gates).

Even though the average area/performance improvements are not too striking, the proposed method is better than [17] in the majority of tested benchmarks (76% in area, 66% in performance).

Note that the final design area/performance strictly depends on the synchronous optimization and mapping processes. We have observed that some circuits are "easier" for LUT mapping than for standard gates mapping (e.g. the *cordic* circuit), and vice versa. This is also one of the reasons for the small summary improvements we have measured – extreme cases exist on both sides. This fact clearly documented in the experimental section.

The proposed method offers a possibility of using *any* synthesis process and it treats the result in the best possible way.

We can conclude that, if the proposed method taken as an *alternative way* of asynchronous logic synthesis, better designs can be achieved.

Let us also mention the *scalability* of the method. The scalability is determined by the scalability of the synchronous logic synthesis tools used, which is considered to be

sufficient even for large industrial designs. The only possible bottleneck introduced is the orthogonalization phase. However, if gates of up to 6 inputs are considered (which is the industrial practice), no significant design time overhead can be expected.

REFERENCES

[1] A.J. Martin et al., The first asynchronous microprocessor: the test results, ACM SIGARCH Comp. Arch. News, vol. 17, no. 4, June 1989, pp. 95-98.

[2] W.J. Bainbridge et al., Delay-Insensitive, Point-to-Point Interconnect using m-of-n Codes, Proceedings of the 9th IEEE Int. Symp. on Asynchronous Circuits and Systems (ASYNC'03), Vancouver, May 2003 , pp. 132-140.

[3] S.H. Unger, S.H., Asynchronous Sequential Switching Circuits, John Wiley & Sons, Inc., 1969

[4] S.M. Nowick, Automatic Synthesis of Burst-Mode Asynchronous Controllers, Ph.D. thesis, Stanfort University, March 1993.

[5] S.M. Nowick D.L. Dill, Exact Two-Level Minimization of Hazard-Free Logic with Multiple-Input Changes, *IEEE CAD*, vol. 14, August 1995, pp. 986-997.

[6] D. Kung, Hazard-Non-Increasing Gate–Level Optimization Algorithm, IEEE Int. Conf. On Computer–Aided Design, 1992, pp. 631-634.

[7] P. Beerel, K.Y. Yun, W.C. Chou, Opimizing Average-Case Delay in Technology Mapping of Burst-Mode Circuits, IEEE Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, 1996, pp. 244-259.

[8] P. Siegel, G.D. Micheli, D. Dill, Automatic Technology Mapping for Generalized Fundamental Mode Asynchronous Designs, IEEE Design Automation Conference, 1993, pp. 61-67.

[9] W.J. Dally, J.W. Poulton, Digital Systems Engineering, Cambridge University Press, 1998.

[10] C.L. Seitz, System Timing, Introduction to VLSI Systems, C. Mead, L. Conway, Addison—Welsey Publishing Company, 1980, pp. 218-262.

[11] C.D Nielsen, Evaluation of Function Block Designs, Technical Report 1994-135, Department of Computer Science, Technical University of Denmark, Denmark, 43, pp. , 1994.

[12] W.B. Toms, D.A. Edwards, Prime Indicants: A Synthesis Method for Indicating Combinational Logic Blocks, Proc. 15th IEEE International Symposium on Asynchronous Circuits and Systems, Async 2009, Chapel Hill, North Carolina, 17-20, May 2009, pp. 139-150.

[13] E.J. Sparsø, J. Staunstrup, M. Dantzer-Sørensen, Design of delay insensitive circuits using multi-ring structures, In Proc. of the European Design Automation Conference (EURO-DAC'92), 1992, pp. 15-20.

[14] M. Ligthart, K. Fant, R. Smith, A. Taubin, A. Kondratyev, Asynchronous Design Using Commercial HDL Synthesis Tools, 6-th Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, 2000, pp. 114-125.

[15] S.C. Smith, J. Di, Designing Asynchronous Circuits using NULL Convention Logic (NCL), Morgan & Claypool, 2009, 96 p.

[16] J. Cortadella, A. Kondratyev, L. Lavagno, C.P. Sotiriou, Desynchronization: Synthesis of Asynchronous Circuits from Synchronous Specifications, IEEE Trans. on CAD, vol.25, No.10, October 2006, pp. 1904-1921.

[17] J. Cortadella, A. Kondratyev, L. Lavagno, C.P. Sotiriou, Coping with the Variability of Combinational Logic Delays, IEEE Int. Conf. On Computer Design, 2004, pp. 505-508.

[18] I. Lemberski, P. Fišer, Asynchronous Two-Level Logic of Reduced Cost, IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, April 15-17, 2009, Liberec, Czech Republic, pp. 68-73.

[19] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification'. http://www.eecs.berkeley.edu/~alanmi/abc/.

[20] R.K. Brayton et al. *Logic minimization algorithms for VLSI synthesis*, Boston, MA, Kluwer Academic Publishers, 1984.

[21] A. Bernasconi, V. Ciriani, F. Luccio, and L. Pagli, A New Heuristic for DSOP Minimization, Proc. 8th Int. Workshop on Boolean Problems (IWSBP'08), Freiberg, Germany, 18.-19.9.2008, pp. 169-174.

[22] Berkeley Logic Interchange Format, University of California, Brekeley, Tech. report, 2005.

[23] S. Yang, Logic Synthesis and Optimization Benchmarks User Guide, Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC.

[24] F. Brglez and H. Fujiwara:,A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan, Proc. of ISCAS 1985, pp. 663-698.

[25] F. Brglez, D. Bryan, and H. Kozminski, Combinational Profiles of Sequential Benchmark Circuits, Proc. of ISCAS, pp. 1929-1934, 1989.

[26] I. Sutherland et al., Logical Efforts: Designing Fast CMOS Circuits, Morgan Kaufmann, 1999, 239 pp.

[27] E.J. Sparsø. S. Furber, *Principles of Asynchronous Circuit Design*, Kluwer Academic Publishers, 2001.

[28] H. Gao, Y. Yang., X. Ma, and G. Dong, Analysis of the effect of LUT size on FPGA area and delay using theoretical derivations, Proc. of the Sixth International Symposium on Quality of Electronic Design", 21.-23. 3, pp. 370-374.

[29] V.I. Varshavski, Self - Timed Control of Concurrent Processes, Kluwer Academic Publisher, 1990.

[30] A. Mokhov, D. Sokolov and A. Yakovlev, Completion Detection Optimisation, Technical Report Series NCL-EECE–MSD–TR–2005-109, 2005, 6 pp.

[31] Y. Zhou, D. Sokolov and A. Yakovlev, Cost-Aware Synthesis of Asynchronous Circuits Based on Partial Acknowledgement, Proc. ICCAD'06, San Jose, November 2006, pp. 158-163.

TABLE III. SUMMARY COMPARISON – AREA (NUMBER OF TRANSISTORS)

k	Proposed	Simple gates [17]		DIMS [13]		Direct logic [11]		NCL [15]	
	Transistors	*Transistors*	*Avg. diff.*	*Transistors*	*Avg. diff.*	*Transistors*	*Avg. diff.*	*Transistors*	*Avg. diff.*
2	3,191,136	3,235,712	2%	2,896,824	-18%	2,655,422	-29%	2,551,032	-34%
3	2,271,142	2,504,552	6%	3,871,184	35%	2,210,690	-14%	N/A	N/A
4	2,191,972	2,218,362	2%	6,122,136	59%	2,326,294	-6%	N/A	N/A
5	2,349,356	2,160,442	-5%	10,191,848	73%	2,747,548	2%	N/A	N/A
6	2,615,018	2,115,466	-14%	17,309,296	80%	3,470,240	12%	N/A	N/A

TABLE IV. SUMMARY COMPARISON – LOGICAL EFFORT

k	Proposed	Simple gates [17]	Average difference
2	6,546	6,586	0%
3	5,376	5,442	0.9%
4	4,955	5,214	3.8%
5	4,952	5,168	3.1%
6	4,933	5,123	2.6%

TABLE V. DETAIED RESULTS, K = 3, AREA

			Synch.	Proposed asynchronous				Simple gates [17]			
	n	*q*	*Trans.*	*Funct. trans.*	*CD trans.*	*Total trans.*	*Overh.*	*Funct. trans.*	*CD trans.*	*Total trans.*	*Diff.*
apex2	39	3	11,550	23,168	22,604	45,772	74.8%	26,040	30,836	56,876	19.5%
c6288	32	32	13,112	22,742	9,212	31,954	59.0%	21,896	26,444	48,340	33.9%
des	256	245	14,866	30,358	24,124	54,482	72.7%	25,160	35,956	61,116	10.9%
mainpla	27	54	19,334	37,646	24,700	62,346	69.0%	26,468	36,452	62,920	0.9%
misex3	14	14	13,094	25,930	21,716	47,646	72.5%	28,164	34,372	62,536	23.8%
prom1	9	40	33,836	61,152	42,364	103,516	67.3%	43,308	57,356	100,664	-2.8%
s35932	1763	2048	41,256	82,654	61,988	144,642	71.5%	58,000	79,804	137,804	-5.0%
s38417	1664	1742	37,480	74,606	65,548	140,154	73.3%	62,340	103,292	165,632	15.4%
s38584.1	1464	1730	44,444	90,704	72,484	163,188	72.8%	76,060	118,916	194,976	16.3%
too_large	38	3	16,236	32,676	31,252	63,928	74.6%	29,164	36,940	66,104	3.3%
rd84	8	4	2,368	4,326	3,508	7,834	69.8%	5,632	6,124	11,756	33.4%
c6288	32	32	13,112	22,742	9,212	31,954	59.0%	21,896	26,444	48,340	33.9%
shift	19	16	554	1,100	796	1,896	70.8%	1,132	1,892	3,024	37.3%
vg2	25	8	436	882	1,004	1,886	76.9%	1,248	1,804	3,052	38.2%
z4ml	7	4	194	364	220	584	66.8%	484	644	1,128	48.2%
ex4p	128	28	3,726	7,500	6,940	14,440	74.2%	12,336	15,820	28,156	48.7%
t481	16	1	2,142	4,294	3,524	7,818	72.6%	7,648	10,084	17,732	55.9%
cordic	23	2	1,808	3,396	3,028	6,424	71.9%	17,036	21,916	38,952	83.5%
Average:							**74.1%**				**6.1%**

TABLE VI. DETAIED RESULTS, K = 3, PERFORMANCE

			Synch.	Proposed asynchronous			Simple gates [17]			
	n	*q*	*Logical effort*	*Funct. logical effort*	*CD levels*	*Total logical effort*	*Funct. logical effort*	*CD levels*	*Total logical effort*	*Diff.*
apex2	39	3	22.0	23.7	9	29.3	19.0	7	26.3	-11.4%
c6288	32	32	128.0	132.7	10	136.3	140.3	10	144.3	5.5%
des	256	245	26.3	25.7	6	34.3	25.0	6	34.3	0.0%
mainpla	27	54	35.0	39.3	9	45.7	43.7	8	51.0	10.5%
misex3	14	14	20.3	21.0	7	29.0	17.7	7	26.0	-11.5%
prom1	9	40	23.3	24.3	7	33.0	26.7	10	35.7	7.5%
s35932	1763	2048	16.7	17.0	7	29.0	18.3	8	32.0	9.4%
s38417	1664	1742	33.7	35.3	7	42.0	31.7	7	42.7	1.6%
s38584.1	1464	1730	34.7	36.7	9	40.3	31.7	8	37.3	-8.0%
too_large	38	3	25.3	26.3	8	31.3	22.3	7	29.0	-8.0%
rd84	8	4	20.3	20.0	5	25.3	14.3	5	21.7	-16.9%
c6288	32	32	128.0	132.7	10	136.3	140.3	10	144.3	5.5%
shift	19	16	8.0	8.0	5	15.7	10.7	4	19.7	20.3%
vg2	25	8	12.0	13.3	4	19.3	12.0	4	18.3	-5.5%
z4ml	7	4	10.0	10.7	2	15.7	12.0	3	17.7	11.3%
ex4p	128	28	17.3	20.3	5	25.7	14.3	7	22.0	-16.7%
t481	16	1	22.3	24.3	4	28.0	20.0	5	23.7	-18.3%
cordic	23	2	23.7	23.0	6	27.7	19.3	7	25.7	-7.8%
Average:										**0.9%**

TABLE VII. DETAIED RESULTS, K = 4, AREA

			Synch.	Proposed asynchronous				Simple gates [17]			
	n	*q*	*Trans.*	*Funct. trans.*	*CD trans.*	*Total trans.*	*Overh.*	*Funct. trans.*	*CD trans.*	*Total trans.*	*Diff.*
apex2	39	3	12,592	27,324	14,740	42,064	70.1%	22,092	22,548	44,640	5.8%
c6288	32	32	20,134	34,116	5,860	39,976	49.6%	22,640	22,532	45,172	11.5%
des	256	245	23,522	45,990	16,596	62,586	62.4%	28,316	30,100	58,416	-7.1%
mainpla	27	54	19,538	40,540	15,876	56,416	65.4%	26,252	31,748	58,000	2.7%
misex3	14	14	15,598	32,520	15,604	48,124	67.6%	27,516	26,596	54,112	11.1%
prom1	9	40	42,766	73,850	26,364	100,214	57.3%	42,692	49,180	91,872	-9.1%
s35932	1763	2048	40,104	75,900	48,156	124,056	67.7%	58,000	70,940	128,940	3.8%
s38417	1664	1742	47,116	100,684	48,668	149,352	68.5%	61,236	87,588	148,824	-0.4%
s38584.1	1464	1730	49,136	105,124	52,356	157,480	68.8%	75,592	103,748	179,340	12.2%
too_large	38	3	17,366	36,616	21,948	58,564	70.3%	28,464	27,588	56,052	-4.5%
cordic	23	2	1,630	3,390	1,564	4,954	67.1%	3,372	3,700	7,072	29.9%
shift	19	16	598	1,264	708	1,972	69.7%	1,132	1,684	2,816	30.0%
cht	15	11	592	1,256	900	2,156	72.5%	1,300	1,812	3,112	30.7%
c8	28	18	498	1,066	676	1,742	71.4%	1,088	1,468	2,556	31.8%
mux	21	1	206	428	356	784	73.7%	516	644	1,160	32.4%
b12	15	9	258	542	380	922	72.0%	644	860	1,504	38.7%
f51m	8	8	412	836	284	1,120	63.2%	1,092	1,172	2,264	50.5%
vg2	25	8	354	848	628	1,476	76.0%	1,632	1,924	3,556	58.5%
Average:							**69.5%**				**1.5%**

TABLE VIII. DETAIED RESULTS, K = 4, PERFORMANCE

			Synch.	Proposed asynchronous			Simple gates [17]			
	n	*q*	*Logical effort*	*Funct. logical effort*	*CD levels*	*Total logical effort*	*Funct. logical effort*	*CD levels*	*Total logical effort*	*Diff.*
apex2	39	3	20.7	22.3	4	28.0	21.7	4	27.3	-2.4%
c6288	32	32	126.7	136.7	4	140.3	134.3	7	138.0	-1.7%
des	256	245	25.0	25.0	5	33.0	21.3	5	30.3	-8.8%
mainpla	27	54	27.0	31.7	5	38.3	45.3	5	49.7	22.8%
misex3	14	14	19.0	20.7	4	27.7	18.3	7	24.7	-12.2%
prom1	9	40	21.3	24.3	7	31.3	27.3	6	33.7	6.9%
s35932	1763	2048	12.7	13.3	6	23.0	18.3	7	30.0	23.3%
s38417	1664	1742	29.3	31.3	6	41.0	33.0	6	40.7	-0.8%
s38584.1	1464	1730	28.3	30.7	6	34.3	33.3	5	37.0	7.2%
too_large	38	3	20.7	22.3	5	27.7	21.7	6	26.3	-5.1%
cordic	23	2	20.7	25.0	4	28.7	21.7	4	26.7	-7.5%
shift	19	16	8.3	10.0	3	15.7	10.7	3	18.0	13.0%
cht	15	11	5.7	6.0	2	12.7	7.0	4	14.0	9.5%
c8	28	18	8.7	10.0	3	14.3	10.3	3	16.0	10.4%
mux	21	1	11.3	13.0	3	16.7	11.0	2	14.7	-13.6%
b12	15	9	6.3	7.0	2	12.7	11.0	3	15.0	15.6%
f51m	8	8	10.0	11.0	3	16.0	12.3	3	18.0	11.1%
vg2	25	8	12.0	15.0	5	19.0	12.3	3	17.3	-9.6%
Average:										**3.8%**