A Complete Synthesis Method for Block-Level Relaxation in Self-Timed Datapaths

W. B. Toms, D. A. Edwards
School of Computer Science, University of Manchester
{tomsw, doug}@cs.man.ac.uk

Abstract

Self-timed circuits present an attractive solution to the problem of process variation. However, implementing self-timed combinational logic can be complex and expensive. This paper presents a complete synthesis flow that generates self-timed combinational networks from conventional Boolean networks. The Boolean network is partitioned into small function blocks which are then synthesised using self-timed techniques. The procedure employs relaxation optimisations to distribute the overheads associated with self-timed networks between function-blocks. Relaxation is incorporated into the function block synthesis procedures, meaning the optimisations can be applied at a much finer granularity than previously possible. The new techniques are demonstrated on a range of benchmarks showing average reduction of 5% *in area, 26% in latency and 48% in energy over gate-level relaxation techniques and 17% in area, 8% in latency and 20% in energy consumption over other block-level relaxation techniques.*

1. Introduction

Process variation is the major challenge currently facing the VLSI industry. In deep sub-micron technologies, timing closure for synchronous systems, which are already clocked at up to 50% below their ideal potential [1], becomes complex. Self-timed circuits [11], whose operation is independent of any external timing reference, are increasingly being seen as a solution to the problems of timing closure in highly variable technologies. The robust timing models employed by these circuits make them extremely tolerant to variations in the propagation delays of circuit components. However, the lack of assumptions about the environment and circuit components make self-timed circuits difficult to specify, create and test. In particular, self-timed combinational logic operations are complex because the validity of an operand needs to be encoded within the data itself. The cost of encoding the datapath in this manner is significant: each data word must be transmitted explicitly and, because the logic level of each wire no longer specifies a data value, data wires must transition into a known (spacer) state in between every transmission. Furthermore, additional logic is required to *indicate* the internal signals of a circuit - to ensure they are in a steady state by the time the outputs are generated.

This paper describes a complete synthesis flow for self-timed circuits which converts conventional Boolean combinational logic netlists into self-timed combinational netlists. The gates of the combinational netlist are combined into function-blocks which are synthesised using self-timed synthesis algorithms. The cost of implementations are reduced by incorporating relaxation optimisations [2], where indication is distributed across function-blocks which share common inputs. Unlike other function-block based methods [3], the approaches in this paper are fully automated. Furthermore, as relaxation techniques are incorporated *within* the self-timed synthesis procedures, relaxation can be applied at a much finer granularity than was previously possible. The system demonstrates significant improvements in area, performance and energy compared to other gate-level and block-level techniques over a range of benchmarks.

2. Indication

Self-timed combinational logic has been the subject of much research, leading to several different terms being applied to the same principles. The work in this paper is based on a model developed specifically for combinational logic circuits by Varshavsky [13]. This model is useful as it defines the properties of self-timed circuits in terms of Boolean equations. The process described by Varshavsky as *indication* is equivalent to *acknowledgement* [15] and *completeness* [6] used subsequently by others. An indicating circuit is equivalent to a *timing-robust* [2] or *input-complete* [5] circuit.

2.1 Definitions

- A multi-valued *variable* v_i can take on symbolic values from $P_i = \{\alpha_0, \alpha_1, \ldots, \alpha_{|P_i|-1}\}$. Each symbolic value maps on to a unique integer $P_i = \{0, 1, \ldots, |P_i| - 1\}$. A binary variable is one in which $P_i = \{0, 1\}$.

- A function, *f*, of *n* variables is a mapping $f: P_1 \times \ldots \times P_n \rightarrow P_f$. In a *Boolean* function $P_f = \{0, 1, *\}$.
- Each element in the domain of function *f* is called a *minterm*.
- In a Boolean function, *f*, the set of minterms for which $f = 1$ is called the *on-set*, the set for which $f = 0$ is called the *off-set* and the set for which $f = *$ is called the *don't-care set*.
- A multi-valued *literal* is a binary valued logic function of the form:

$$v_i^S = \begin{cases} 0 & if \quad v_i \in S \\ 1 & otherwise \end{cases}$$

  where $S \subseteq P_i$. If v_i is a binary variable then $v^{\{1\}}$ is written as v_i, $v^{\{0\}}$ is written as $\overline{v_i}$ and $v^{\{0, 1\}}$ is written as v_i^*.
- A *product term* is a Boolean product (AND) of literals. A *cube* is the set of minterms which can be described by a product term.
- Product *y* contains product *x* ($x \subseteq y$) if the cube of *x* is a subset of the cube of *y*.
- An *implicant* of a (*boolean*) function is a product term which contains no minterms of the function's off-set.
- A *prime implicant* is an implicant contained in no other implicant of the function.
- A *cover* of a function is a set of implicants which contains all of the minterms of the on-set and no minterms of the off-set.

2.2 Delay-Insensitive Encoding

As there are no external timing references in self-timed circuits, the validity of data must be encoded within the data itself using a Delay-Insensitive (or *unordered*) code [14]. In a DI-code, no code word is contained within any other, allowing the arrival of a valid data word to be determined unambiguously. The most common form of DI-code is *dual-rail*, where each binary bit is encoded with two wires and a transition on one of the two wires signals the arrival of data. Large datapaths are formed by concatenating code-groups together. The dual-rail code can be generalised to *m-of-n* codes where data words are signified by *m* transitions on *n* wires. In *m*-of-*n* encodings the logic-level of each wire only signifies the presence or absences of data. Therefore, in most self-timed systems, all wires must transition to a known (*spacer*) state in between data transmissions. In this paper we only consider Return-to-Zero encodings, where all wires must transition to zero in the spacer state.

2.3 Allowed-Transition Sets

The behaviour of an indicating circuit is defined by sets of transitions on the inputs (or outputs) of the circuit called *Allowed-Transition Sets* (ATS). Each ATS describes a complete transition from a valid data value to a valid spacer value (or vice versa) on a set of circuit variables. An ATS consists of a set of transitions on individual variables which may occur in any order. The concept of an ATS is similar to that of a Multiple-Input Change (MIC) in burst mode circuits [8], and (like MICs) each ATS, (*a*-*b*), has an associated transition cube [*a*,*b*] which contains all the possible states of the variables (minterms) that may be reached between *a* and *b*. Also associated with each ATS is a product called the *transition variation term* $\varepsilon(a, b)$ which represents the final values of any variables that transition during the ATS. The set *N*(*a*,*b*) of ATS (*a*-*b*) contains all the minterms, a^i, of [*a*,*b*] which are *adjacent* to *b* (have a Hamming distance of 1). As the transitions in an ATS may occur in any order, the number of elements in *N*(*a*,*b*) is equivalent to the number of variables that transition in the ATS (*a*-*b*).

A combinational logic circuit consists of two sets of variables: A set of *n* inputs, $X = \{x_1 \ldots, x_n\}$, and a set of *m* outputs, $Y = \{y_1, \ldots, y_m\}$. An ATS on the circuit inputs, *X*, causes a subsequent ATS on the circuit outputs, *Y*. The behaviour of the function-block is defined by two multi-valued functions:

- $Y = F(X)$ which maps the data values of X (D^X) to data values of Y (D^Y).
- $Y = G(X)$ which maps the spacer values of X (S^X) to spacer values of Y (S^Y).

The encodings used in this paper have only a single spacer (the zero value ($\{v_1 = 0, \ldots, v_n = 0\}$) and so $G(s^X) = s^Y$ for all function blocks.

In order to construct a physical circuit for a function block, the multi-valued functions are implemented by a *system of inherent functions* (SIF). In an SIF each output of the function block is determined by a binary cover function:

$$y_i = f_i(x_1, \ldots, x_n) \quad 1 \le i \le m$$

Each f_i is the *encoded function* of variable y_i formed by mapping the multi-valued functions $Y = F(X)$ and $Y = G(X)$ to the binary encoding of code system *X*.

Example 2.1 *A* is a function block with two dual-rail encoded binary inputs, it has four input variables $X = \{x_1, x_2, x_3, x_4\}$. The encoding of the inputs means there are four possible data values and a single spacer:

$$D^X = \{d_1^X = 1010, d_2^X = 0110, d_3^X = 1001, d_4^X = 0101\}$$

$$S^X = \{s^X = 0000\}$$

Code system *X* contains eight ATS:

$$(s^X - d_1^X), (s^X - d_2^X), (s^X - d_3^X), (s^X - d_4^X)$$

$$(d_1^X - s^X), (d_2^X - s^X), (d_3^X - s^X), (d_4^X - s^X)$$

ATS $(s^X - d_1^X)$ has the transition cube:

$$[s, d_1^X] = \overline{x_2}\,\overline{x_4} = \{0000, 1000, 0010, 1010\}$$

the transition variation term:

$$\varepsilon(s^X, d_1^X) = x_1 x_3$$

and the adjacent transitions

$$N(s^X, d_1^X) = \{1000, 0010\}$$

There are eight possible ATS:

$$(s - d_1), (s - d_2), (s - d_3), (s - d_4)$$
$$(d_1 - s), (d_2 - s), (d_3 - s), (d_4 - s)$$

The output of the function block is a single dual-rail signal. The encoding of the outputs means there are two data values and a single spacer:

$$D^Y = \{d_1^Y = 10, d_2^Y = 01\}$$

$$S^Y = \{s^Y = 00\}$$

Code system Y contains four ATS:

$$(s^Y - d_1^Y), (s^Y - d_2^Y), (d_1^Y - s^Y), (d_2^Y - s^Y)$$

For transition $(s^Y - d_1^Y)$:

$$\varepsilon(s^Y, d_1^Y) = y_1$$

and

$$N(s^Y, d_1^Y) = \{00\}$$

$F(X)$ is defined as follows:

$$d_1^Y = d_1^X + d_2^X + d_3^X$$

$$d_2^Y = d_4^X.$$

2.4 Indication

In order to eliminate timing assumptions in self-timed circuits the outputs of a circuit are used to indicate to the environment that the internal gates of a circuit are in a steady state and the circuit is ready to accept more input. Varshavsky defines the *indication of the inputs* (and internal signals) of a circuit in terms of the *translation of individual transitions* to the outputs.

An input transition is *translated* to the output of a function if its arrival causes the output to transition. Within each ATS not all input transitions are directly translated to the output of a function, as this would mean the function must change value after every input transition. However, all input transitions must be *capable* of causing an output transition if they are the last to occur. This property allows the function to indicate all input transitions regardless of the order of their arrival.

Boolean differences of functions are used to determine whether a function translates all of its input transitions. The Boolean difference of function $f_j(x_1, \ldots, x_n)$ with respect to variable x_i is given by:

$$\frac{\partial f_j}{\partial x_i} = f_{jx_i} \oplus f_{j\bar{x}_i}$$

where f_{jx_i} and $f_{j\bar{x}_i}$ are the *cofactors* of x_i in f_j (the functions obtained by replacing all occurrences of x_i in f_j with a 1 or 0 respectively). Boolean differences are used to determine the conditions under which a function is dependent (or independent) on a variable and are commonly used to generate input vectors for stuck-at-fault testing.

Definition 2.1 *An output, y_j,* ***translates*** *an input transition on input x_i in the ATS (a-b), if for the adjacent transition, (a^i-b), where* $a^i \in N(a, b)$ and $\varepsilon(a, b) = x_i$, *the values of the variables in* $[a^i, b]$ *form a solution to the equation:*

$$\frac{\partial f_j}{\partial x_i} = 1$$

Example 2.2 For the function block from example 2.1, a possible implementation for the cover function variable y_1 is:

$$y_1 = x_1x_3 + x_2x_3 + x_1x_4$$

The transition $(s^X - d_1^X)$ has the transition constant term:

$$\varepsilon(s^X, d_1^X) = x_1x_3$$

and the adjacent combinations:

$$N(s^X, d_1^X) = \{n^1 = 1000, n^2 = 0010\}$$

Considering the adjacent transition $(n^1 - d_1^X)$, the transition on x_3 is translated by y_1 as the value of the variables in the transition cube form a solution for the equation generated from the Boolean difference of the variable in the variation term:

$$\varepsilon(n^1, b) = x_3 \quad \text{and} \quad [n^1, b] = x_1\overline{x_2}\,\overline{x_4}$$

$$\frac{\partial f_j}{\partial x_3} = x_1 \oplus x_1x_4 = 1 \oplus 0 = 1$$

Therefore, function y_1 indicates the transition x_3 for the ATS $(s^X - d_1^X)$.

The concept of translation is used to define the indication of input variables and circuits:

Definition 2.2 *An input variable, x_i, is* ***indicated*** *if for each ATS, (a-b), in which x_i transitions there exists an output y_i of the circuit which translates the transition on x_i.*

Definition 2.3 *A circuit is* ***indicating*** *if all its input variables are indicated.*

2.5 Canonical Architecture

To determine whether a particular circuit implementation is indicating requires solving Boolean difference equations for every input transition in all the ATS of the circuit. These checks add a significant overhead to the synthesis process and so it is preferable to implement circuits in a canonical architecture in which the indication of the circuit is guaranteed. In the canonical architecture, each function is constructed from the sum of a set of Muller C-elements [7]. Each C-element implements two ATS, a spacer to data transition (s-d_k) and the corresponding spacer to data (d_k-s) transition. In RTZ systems the inputs to the C-element correspond to those variables in the transition variation term of the (s-d_k) ATS.

As the inputs of the circuit are encoded within a DI-code, no transition variation term is contained within any other and each C-element is mutually-exclusive in the normal behav-

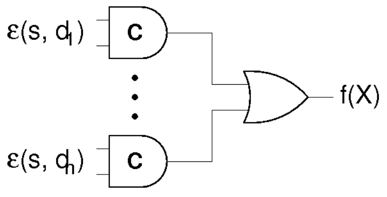

Figure 1: Canonical Architecture

iour of the circuit. For a given ATS, only one C-element of a function transitions and the output is dependent on all of the inputs to the C-element. Therefore, provided that there is a C-element implementing each ATS in some function of the circuit, the circuit is indicating. This canonical architecture forms the basis of many indicating circuit styles such as DIMS [10] and NCL-D [6]. However, because each C-element has an input for each transition within an ATS, the size of the canonical architecture is large and, as each output must wait for all of the input transitions before transitioning, the architecture has worst-case performance.

3. Indicating Combinational Logic Synthesis

3.1 Desynchronisation

In order to create an indicating circuit, all of the ATS must be fully enumerated to ensure that each input transition is translated by an output transition. For large datapath components this becomes infeasible and so a method of decomposing datapath components must be employed. One of the most popular methods of constructing indicating datapath components is *desynchronisation* [1][5] where conventional synthesis tools are used to synthesise a gate-level network which is then converted by expanding each gate into an indicating *gate-operator* circuit implementing the same function. In order to encode data validity, each binary variable in the design is expanded into a dual-rail pair of variables. In the NCL-D design flow [6], figure 2.*i*, each gate-operator is implemented using the canonical architecture. However, because of the overheads associated with the canonical architecture, several optimisations have been developed.

In the NCL-X design flow [5], figure 2.*ii*, each binary variable is replaced by 3 variables, which represent a dual-rail pair and an additional *valid* signal which translates the transitions of the dual rail pair. Employing a valid signal means translation does not need to be performed within the gate-operators and can be implemented using an external *completion detector.* This greatly simplifies the implementation of the gate-operators and improves performance as the completion detection can be done in parallel with logic. However, the area and energy consumption of these circuits is often increased due to the addition of a valid signal for each variable in the design and the extra completion detection.

3.2 Gate-level Relaxation

An alternative approach to optimisation of desynchronised datapaths was suggested concurrently by Jeong [2] and Zhou [15]. Both of these methods rely on the *relaxation* of the indication of inputs to individual gate functions. In the NCL-D approach each gate operator indicates the transitions on all of its inputs, meaning that the transitions of nodes with multiple fan-out are indicated by all of the gates in its fan-out. Definition 2.2 states that only one output is required to translate each input transition and so the indication of all but one gate operator in the fan-out of a variable can be *relaxed*. The aim of both approaches is to determine a distribution of the indication of each variable to reduce the total cost (using various metrics) of implementing the network. A set of implementations of each gate operator is created for all the possible relaxation combinations of the inputs. Figure 3 shows the relaxed implementations of a Dual-Rail AND gate. The arrows determine whether the implementation indicates the spacer-to-data transitions (↑) or the data-to-spacer transitions (↓) (there are only 5 distinct implementations as many of the implementations are equivalent). UCP [2] or ILP [15] solvers are then used to select an implemen-

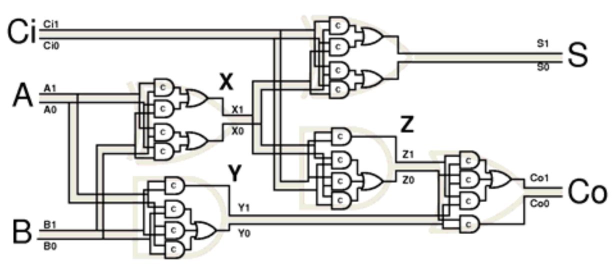
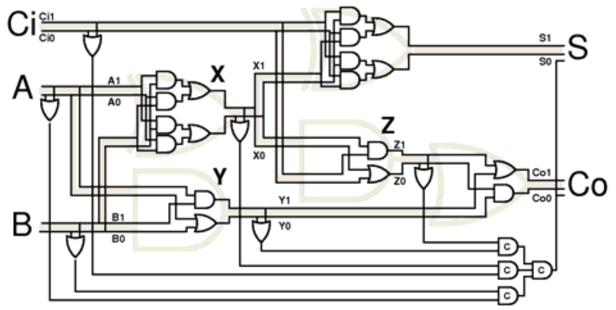
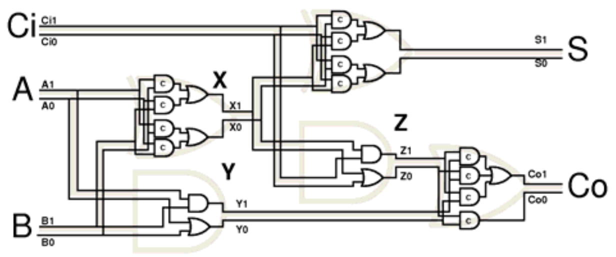

Figure 2: Desynchronisation Implementations of Ripple Carry Adder: *i*) NCL-D, *ii*) NCL-X, *iii*) Gate-level Relaxation

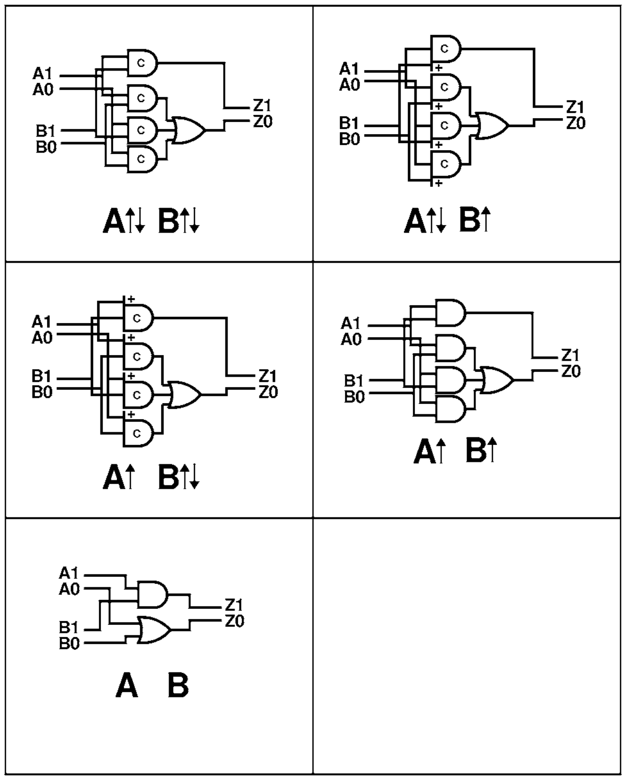

Figure 3: Relaxed Implementations of Dual-Rail AND gate.

tation for each gate that will minimise the cost yet maintain the indication of the network.

In the full-adder circuit in figure 2.*iii*, inputs A and B are both connected to two gates, an XOR (driving signal X) and an AND gate (driving signal Y). Therefore, it is not necessary for both gates to indicate the transitions on both signals and the implementations of the gates may be relaxed. In this case, cost of the implementation is minimised by employing a fully relaxed implementation for the AND gate and a fully-indicating implementation for the XOR gate. The same optimisation occurs on the gates driving Z and S. As the signals Y and Z have only a single fanout, the OR-gate driving Co cannot be relaxed.

Despite the dramatic improvements of the optimisations over the original NCL-D circuits, there are still problems inherent in the desynchronisation design-flow. The cost functions associated with gates in a conventional tool flow are different from the costs associated with their equivalent indicating gate operators. Implementing dual-rail signalling at the granularity of individual gate operators increases significantly the switching overheads of indicating circuits, this is further increased by the addition of a validity signal in the NCL-X flow. Finally, the one-to-one substitution process of gates to gate-operators within the desynchronisation flow makes employing more complex, lower power, m-of-n encodings very difficult to apply.

3.3 Block-level Relaxation

In order to overcome some of the problems inherent in desynchronisation, a block-level approach was proposed by Jeong [3]. Here, datapaths are composed from function-blocks consisting of multiple (< 10) binary inputs and outputs, which are connected by DI-encoded channels. The block-level approach can reduce the cost of gate-level implementations by allowing the indication of inputs to be shared between the outputs of the blocks. Furthermore, as each block corresponds to multiple gates, the amount of DI signalling between blocks is reduced, and more complex encodings may be used.

Jeong [3] extended gate-level relaxation algorithms to the block-level. However, the approach described was largely manual. In order to construct a block-level implementation, designs needed to be decomposed manually and implementations for each block provided by the designer. As in gate-level relaxation, the cost of the network implementation was reduced by selecting relaxed implementations of certain blocks without violating the indication of the network. Therefore, the designer needed to create multiple implementations of each block for the selection procedure. To reduce the design effort, the selection algorithm could only select between fully indicating and fully-relaxed implementations. Furthermore, although several potential strategies for reducing the cost of indicating implementations were outlined, no method of constructing arbitrary indicating blocks was presented.

This paper presents a fully automated block-level synthesis system, that creates a block-level indicating network from a gate-level Boolean network. The gates of the original network are clustered to form function-blocks, which are then synthesised using indicating synthesis methods. Relaxation optimisations are incorporated in to the system by modifying the synthesis methods to synthesise partially indicating function-blocks. The structure of the synthesis method allows the cost of indication of individual inputs to be quantified and so partially relaxed implementations may be selected without the need to create all possible implementations of each function block. The results show significant improvements in performance, area and energy consumption over other gate-level and block-level techniques.

4. Prime Indicant Synthesis

In [12] a synthesis method was presented to synthesise low cost indicating implementations of arbitrarily-encoded function-blocks. The method adapts conventional synthesis techniques to determine a low cost two-level Sum-of-Products implementation of each function-block. The work in this paper extends these synthesis methods to produce partially-indicating implementations of function-blocks that are used during the relaxation process. In order to describe how the partially indicating implementations are produced the original procedures are presented in detail.

The nature of indicating logic has a significant impact on the synthesis process:

- To minimise the cost of an implementation, the indication of inputs is distributed between all the functions. Therefore, the optimisation of each function is dependent on the implementation of other functions in the function block.
- As products are *expanded* in the synthesis procedure literals are removed from them, meaning the product no longer translates transitions on these inputs. In order to maintain indication, each function can no longer be constructed as a sum of prime implicants and other (non-prime) implicants need to be considered.

These properties increase the complexity and search space of the synthesis algorithm, as the optimisation of all functions must be executed in parallel and all possible implicants of each function may need to be considered. In order to overcome these difficulties, the synthesis method uses a two stage process. Firstly, the lowest cost *indicant cover* of each function is determined. Then, the untranslated input transitions of the function block are determined, and distributed between the functions of the function block.

4.1 Indicant Cover

The first stage of the synthesis procedure is to determine the minimum cost *indicant cover* for each function in the function block (the cost functions in [12] were based on literal counts). In two level SOP indicating logic, transitions on the output of a function must be used to translate transitions on the products of the function, as well as transitions on the function inputs. An *indicant* is an implicant of a function whose transitions are indicated by that function. As the indicant is a single product, it will also translate all of the transitions on its inputs for any ATS in which it transitions. It was shown in [12] that any minimum cost indicating implementation of a function must always be constructed from a covering of indicants. As described in section 2.5, a SOP implementation can be made indicating by ensuring all of the products are mutually exclusive. The indicant cover procedure therefore determines the lowest cost mutually-exclusive implementation for each function in the function block.

In conventional circuit synthesis, an implementation for a function is generated from the smallest set of prime implicants that *cover* all the minterms of the circuit. In indicating logic, the transition variation terms of each ATS, rather than function minterms, form *required cubes* that must be covered. A mutually-exclusive implementation for a function can be generated by ensuring that the cover is *non-overlapping*, i.e. that no more than one prime-implicant covers each required cube. However, it may not be possible to construct a non-overlapping cover solely from the prime-implicants of a function, and so all function implicants must be considered. As the total number of implicants in a function can be large, determining a minimum cost cover of all of the implicants is infeasible. Therefore, to reduce the complexity of the indicant cover procedure, an optimal prime-implicant covering for the function is generated and then made mutually-exclusive. To make the cover mutually-exclusive, all of the possible sub-implicants of any prime-implicants that overlap are enumerated. A non-overlapping cover for these implicants can then be generated using a modified UCP framework.

Example 4.1 The required cubes of function f_j are:

$$f_j = a_0b_0c_0 + a_0b_0c_1 + a_0b_1c_0 + a_1b_0c_0$$

The minimum cost set of prime-implicants for f_j is

$$PI = \{a_0b_0, a_0c_0, b_0c_0\}$$

As all of these prime-implicants overlap, the sub-implicants of each prime-implicant must be enumerated, giving the total set of implicants as:

$$IMP = \{a_0b_0, a_0b_0c_0, a_0b_0c_1, b_0c_0$$
$$a_1b_0c_0, a_0c_0, a_0b_1c_0\}$$

Figure 4 shows a minimum non-overlapping cover of the implicants which results in the function:

$$f_j = a_0b_0 + a_0b_0c_1 + a_1b_0c_0$$

	a_0b_0	b_0c_0	a_0c_0	$a_0b_0c_0$	$a_0b_1c_0$	$a_0b_0c_1$	$a_1b_0c_0$
$a_0b_0c_0$	1	1	1	1			
$a_0b_1c_0$			1		1		
$a_0b_0c_1$	1					1	
$a_1b_0c_0$		1					1

Figure 4: Non-overlapping cover for f_j

4.2 Indicant Reduction

A minimum cost indicant cover function block implementation will indicate all of the transitions on any internal gates within the function block. It will also translate input transitions on any inputs to the indicants of its functions. However, as the initial required cubes of the function have been expanded, the properties of the canonical architecture no longer hold and the function block is no longer guaranteed to indicate all input transitions. The next stage of the synthesis process is to reduce, by re-applying the expanded literals, a subset of the indicants in the function block to ensure all input transitions are indicated.

As the indicants of a function are mutually-exclusive, each indicant covers a different set of required cubes. Therefore, when an indicant is reduced it must be replaced by two or more indicants which cover the same required cubes as the original indicant. Furthermore, to ensure the resulting function is still indicating the new indicants must also be mutually-exclusive. As the required cubes of indicating functions are unate in all variables [13], it is not possible to reduce an indicant by a single variable and its complement, and different variables must be inserted into each of the new indicants to ensure they are mutually exclusive.

Example 4.2 Indicant $I = a_0b_0$ covers the required cubes:

$$\{a_0b_0c_0d_0, a_0b_0c_0d_1, a_0b_0c_1d_0, a_0b_0c_1d_1\}$$

Reducing I by literal c_0 and c_1 creates two indicants:

$$I_1 = a_0b_0c_0 \text{ and } I_2 = a_0b_0c_1$$

which cover the cubes:

$$\{a_0b_0c_0d_0, a_0b_0c_0d_1\} \text{ and } \{a_0b_0c_1d_0, a_0b_0c_1d_1\}$$

respectively, and hence are mutually exclusive.
When a pair of reductions are applied to an indicant, the number of resultant indicants is multiplied:

Example 4.3 $I = a_0$ and covers the terms:

$$\{a_0b_0c_0, a_0b_0c_1, a_0b_1c_0, a_0b_1c_1\}$$

Reducing I by b_0 (and b_1) results in the indicants:

$$I_1 = a_0b_0 \text{ and } I_2 = a_0b_1$$

and reducing I by $c0$ (and c_1) results in the indicants:

$$I_1 = a_0c_0 \text{ and } I_2 = a_0c_1$$

Applying both reductions together results in four indicants:

$$\{a_0b_0c_0, a_0b_0c_1, a_0b_1c_0, a_0b_1c_1\}$$

In order to minimise the cost to the network, the reductions need to be distributed between the indicants of the network. In [12] a distributed UCP algorithm was presented which multiplies the cost of indicants in the solution if reductions from the same indicant have been previously selected. However, this increases the complexity of finding a solution as conventional reduction techniques (such as dominance [9]) rely on fixed costs and can no longer be employed.

4.3 Off-set Synthesis

In indicating combinational logic circuits, both spacer-to-data and data-to-spacer ATS must be indicated. This is achieved by implementing each required cube using C-elements. Each C-element sequentially composes two required-cubes: an *on-set* cube (which translates spacer-to-data ATS) and an *off-set* cube (which translates data-to-spacer ATS). The off-set cube of a C-element consists of the complement of all of the literals in the on-set cube. The composition of on-set and off-set cubes, means the off-set cubes of a function do not need to be mutually-exclusive and therefore can be further optimised once a fully-indicating indicant (on-set) cover has been created for a function-block. This has the effect of reducing the number of C-elements and decreasing the overall cycle time of the circuit. In [12] a number of strategies were presented to reduce the off-set, that could target different cell library requirements.

Figure 5 shows the prime-indicant synthesis algorithm.

```
prime_indicant_synthesis(functionblock) {

   // Indicant Cover
   foreach function in functionblock {
      ic = minimum_cost_indicant_cover(function);
      foreach requiredcube of f {
         calculate the missing literals in ic
      }
   }
   missing_literals = total missing literals;
   reduced_missing_literals =
         minimum_cost cover of missing literals;
   // Reduction
   substitute reduced_missing_literals into functions;

   offset_literals =
      minimum_cost covering of off-set literals;
}
```

Figure 5: Prime Indicant Synthesis Algorithm

5. Block-Level Relaxation Synthesis

The prime indicant synthesis method is a novel method of synthesising low cost fully-indicating function blocks. However, because all possible ATS must be enumerated, the method is unsuitable for synthesising large datapath structures. In the remainder of this paper we present a complete synthesis method that allows such structures to be synthesised using the prime-indicant approach. As demonstrated by gate-level and block-level relaxation techniques, employing fully-indicating function blocks everywhere in a datapath structure introduces unnecessary overheads. The methods in this paper extend the synthesis procedures to allow partially indicating implementations to be synthesised. By quantifying the cost of indicating individual variables, an automated block-level relaxation process can be created that can relax *individual variables* within function block implementations.

5.1 Clustering

The synthesis method outlined in this paper takes a conventional Boolean combinational logic netlist as input. In order to exploit the block-level synthesis and relaxation techniques the netlist must be restructured into a set of interconnected function blocks. This is achieved by employing a clustering algorithm similar to the type employed when generating PLAs from combinational networks in [4].

The aim of the algorithm is to incorporate as many gates of the initial netlist into each cluster without exceeding a given maximum input size. The output functions of the cluster are then re-specified in terms of the primary inputs of the cluster. During the synthesis procedure the indication of the inputs of a function block will be distributed throughout the outputs of the block. It is possible that functions may become dependent upon variables upon which they were not dependent in the initial network. Therefore, care must be taken when constructing clusters to prevent circular dependencies arising in the network, which may cause deadlock.

The clustering algorithm, shown in figure 6, proceeds from the outputs. Each cluster is initiated from a single variable. To prevent circular dependencies, the source gate of the variable can only be incorporated into a cluster if the variable is either a primary output or all the fan-out gates of the variable are already clustered. Any non-clustered fan-out gates of the variable must be incorporated before the source gate. Once a gate is incorporated its inputs become inputs to the cluster. If all of the gates in the fan-out of a cluster cannot be incorporated into the cluster, then as many fan-out gates as possible will be incorporated. The algorithm proceeds by recursively incorporating the fan-out and source gates of any

```
add_variable_to_cluster(cluster,v) {
   variables_added = 0;
   if(v != primary_output) {
      foreach g in fanout(v) {
         if(g not already clustered) return FALSE;
      }
   }
   foreach g in fanout(v) {
   if((inputcount(cluster) + inputcount(g))≤ MAXINPUTS)
      cluster = cluster ∪ g;
      foreach i in inputs of g {
         if(add_variable_to_cluster(cluster,i))
            variables_added++;
      }
   }}
   if(variables_added > 0) return TRUE;
   else return FALSE;
}

cluster_netlist (Netlist) {
   clusters = ();
   variables = outputs of netlist;
   foreach v in variables {
      new_cluster = ();
      if(add_variable_to_cluster(new_cluster,v)) {
         clusters = clusters ∪ new_cluster;
         variables = variables ∪ inputs of cluster;
      }
   }
   return clusters;
`
```

Figure 6: Clustering Algorithm

new inputs until no further gates can be added without violating the maximum input count.

Example 5.1 Figure 7 shows an example combinational circuit, clustered using a maximum input size of four. The NAND gates driving the outputs each form a single cluster. As each gate shares inputs with the other gates, the clustering algorithm cannot include any of the NOR gates until all of the NAND gates have been clustered. However, because each NAND gate shares only one input with the other two gates, the combined input count of any pair of NAND gates is five, and these gates cannot be clustered together unless the maximum input size is increase.

5.2 Block-Level Relaxation Synthesis

The two stage approach of prime-indicant synthesis was developed to cope with the complexity of synthesising minimal indicating implementations. However, this approach also facilitates the adaptation of the procedure to incorporate relaxation optimisations. The algorithm for block-level relaxation synthesis is shown in figure 8. In the first stage a minimum cost indicant cover of each function block in the network is produced. The relaxation algorithm is then applied to determine which function blocks should indicate which transitions. Finally the indicant reduction procedure is applied to each block (with a reduced set of untranslated transitions). The operation is then repeated to optimise the off-set of functions.

5.2.1 Indicant Cover

A minimum cost implementation of the network, that indicates all of the internal transitions within function blocks, but not necessarily all transitions on the inputs of

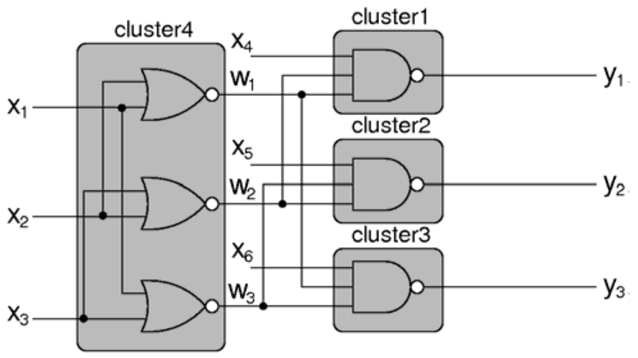

Figure 7: Clustering of Example Circuit

```
block_level_relaxation (Block-Level_Network) {

   // Indicant Cover
   foreach functionblock in Block-Level_Network {
      foreach function in functionblock {
      ic = minimum_cost_indicant_cover(function);
      foreach requiredcube of f {
         calculate the missing literals in ic
      }
   }
      calculate_function_block_missing_variables;
      determine cost of each missing variable;
   }
   // Relaxation
   missing_variables = total missing variables;
   reduced_missing_variables =
         minimum_cost cover of missing variables;

   // Reduction
   foreach functionblock in Block-Level_Network {
      calculate missing literals from
         reduced missing variables;
      substitute reduced_missing_literals in functions
   }

   // Off-set Synthesis
   foreach functionblock in Block-Level_Network {
      missing off-set variables =
            missing on-set variables;
   }
      // Relaxation
   missing_variables = total missing variables;
   reduced_missing_variables =
         minimum_cost cover of missing variables;
      // Reduction
   foreach functionblock in Block-Level_Network {
      calculate missing literals from
         reduced missing variables;
      substitute reduced_missing_literals into
         off-set functions
   }
```

Figure 8: Relaxation Synthesis Algorithm

function blocks, can be created by constructing a minimum cost indicant cover of each function block in the network. In the indicant cover, transitions on the inputs of each indicant are translated by the output function. It is impossible to relax the indication of the indicant covers any further without violating the indication of the internal signals in the function block. Therefore, there is a set of input transitions which are automatically translated by each output function which are removed from the relaxation process.

Example 5.2 Figure 9.*i* shows the canonical architecture

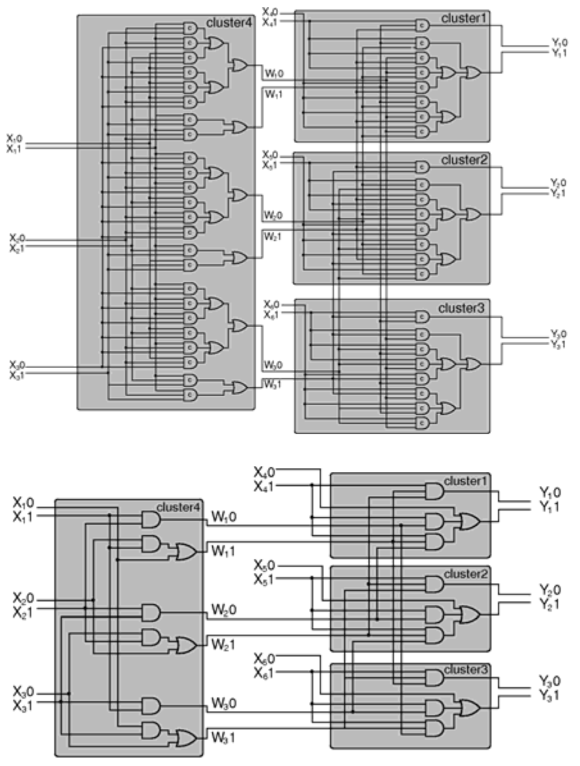

Figure 9: Dual-rail Expansion of Example 5.1. *i*) Canonical Architecture, *ii*) Minimum-Cost Indicant Cover

implementation of the dual-rail expansion of the example circuit shown in figure 8. A minimum cost indicant cover implementation is shown in figure 9.*ii*. It should be noted that the minimum-cost indicant cover implementations of the 2-input NAND gates in cluster 4 are more expensive than the fully-relaxed gate-level AND implementations shown in figure 3, this is because the indicant cover algorithm ensures all indicants are mutually-exclusive (even those that contain a single literal).

5.2.2 Relaxation

Once a minimum cost implementation has been created for each function block, the untranslated transitions of the whole network can easily be calculated. This information can then be used by a reduction selection procedure to distribute the translation between function blocks sharing common inputs. The indicant reduction procedure of prime-indicant synthesis provides an effective method for evaluating the cost of the translation of individual transitions in a function block and a technique to alter the implementation to translate the transitions.

Therefore, the relaxation selection procedure can be used to distribute the translation of *individual transitions* between the function blocks of a network. While this is a very powerful technique, it is infeasible to apply relaxation at this granularity across large combinational logic networks due to the total number of possible transitions. Therefore, the relaxation procedure described in this paper distributes the *indication of variables* between function blocks rather than the translation of transitions.

	Cluster1						Cluster2						Cluster3					
	$X_4 0$	$X_4 1$	$W_1 0$	$W_1 1$	$W_2 0$	$W_2 1$	$X_5 0$	$X_5 1$	$W_2 0$	$W_2 1$	$W_3 0$	$W_3 1$	$X_6 0$	$X_6 1$	$W_1 0$	$W_1 1$	$W_3 0$	$W_3 1$
$X_4 0$	1																	
$X_4 1$		1																
$X_5 0$							1											
$X_5 1$								1										
$X_6 0$													1					
$X_6 1$														1				
$W_1 0$			1												1			
$W_1 1$				1												1		
$W_2 0$					1				1									
$W_2 1$						1				1								
$W_3 0$											1						1	
$W_3 1$												1						1

Figure 10: Covering table for Relaxation Process

As described in section 4.2 the cost of reducing an indicant multiplies when more than one variable is applied, and a distributed unate covering algorithm must be used. During the relaxation process, the cost of indicating a variable within a function block may change depending upon which other variables are also indicated by the function block. To reduce the complexity of the relaxation covering problem, the cost of indicating each unindicated variable is *approximated* by calculating the additional cost of indicating only that variable. A distributed UCP covering algorithm that multiplies the cost of selecting more than one variable in each function block is then used to select which variables will be relaxed by which function blocks. While this only approximates the actual cost of the final implementation, it reduces the need to have individual columns in the UCP table for each possible combination of unindicated variable in each block.

Example 5.3 In the minimum-cost indicant cover figure 9.*ii*, the implementation of cluster 4 indicates all of the transitions on variables $X_1 0$, $X_1 1$, $X_2 0$, $X_2 1$, $X_3 0$ and $X_3 1$, and so these variables are removed from the relaxation process (although as they are not shared by another cluster their indication could not be distributed anyway). None of the remaining clusters indicate their input variables and so the relaxation process distributes the indication of the remaining input (and intermediate) variables between the clusters 1 to 3. The covering table for the relaxation process is shown in figure 10, and a minimum cost covering is highlighted. The cost function of the covering algorithm distributes the indication of the "*W*-variables" to minimise the cost of the final implementation.

5.2.3 Indicant Reduction

The reduction selection procedure determines the set of variables each block must indicate. Each variable is then remapped to a set of transitions within the function-block and the indicant reduction procedure is used to translate the transitions.

Example 5.4 Figure 11.*i* shows the relaxed implementation of the example circuit. The indicants of the clusters 1 to 3

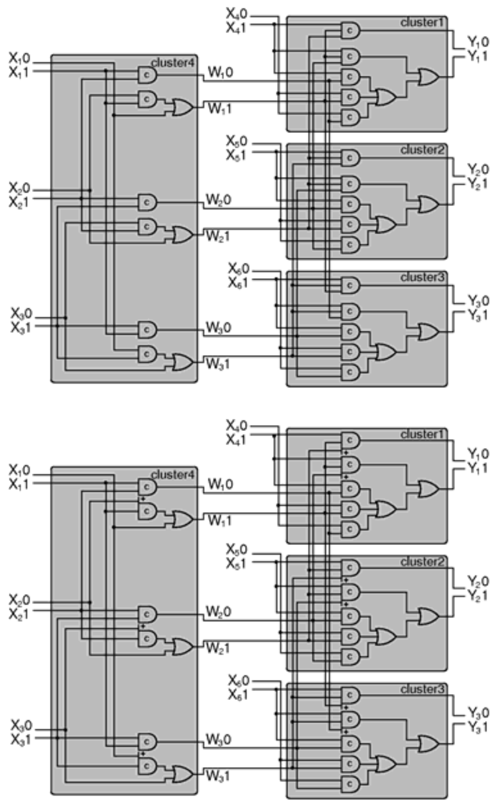

Figure 11: Relaxed Implementations of Example 5.1 *i*) On-set Relaxation, *ii*) Off-set relaxation

have been reduced to indicate the transitions of the inputs as governed by the relaxation procedure. The benefits of allowing partially indicating implementations of clusters can be determined by comparing the implementations of these clusters in figures 9.*i* and 11.*i*. The relaxation methodology proposed by Jeong does not permit partial-indication and, as each of the clusters 1-3 must indicate at least one variable (the relevant "*X*-variable"), fully-indicating implementations would have be employed for all these clusters.

5.2.4 Off-set Synthesis

Relaxation can be used to further optimise the off-set required cubes of the function block once the final on-set implementations have been determined. In the results presented in this paper we restrict indication of the offsets of a function block only to those variables whose on-sets were indicated by the same function block (although it is possible to indicate different variables in the on and offsets of each indicant). Unlike the on-set reduction methods a conventional UCP algorithm for relaxation selection can be used as the cost of reducing the off-set required cubes is not multiplicative.

Example 5.5 Figure 11.*ii* shows an implementation of the example circuit after off-set relaxation. As the relaxation process distributed the indication of the on-set across the clusters, the ability to reduce the off-set is reduced, and only a few generalised C-elements may be employed.

6. Results

The synthesis techniques were demonstrated over a set of ISCAS combinational logic benchmarks. The circuits were targeted at a 180 nm ST Microelectronics technology, using a conventional standard-cell library (C-elements were implemented using complex gates with feedback). Each implementation was place and routed by Cadence Encounter and extracted with lumped single capacitances using Cadence Diva. Simulations were executed in Synopsys Nanosim and the results of each circuit were generated from a sequence of random input vectors. Table 1 shows the area of each implementation, the average latency and the average energy consumption of each input vector.

Each benchmark was implemented using the gate-level relaxation algorithms of Zhou [15] as well as several function block level implementations:

- Fully indicating synthesis without relaxation
- Jeong block-level relaxation: the relaxation technique proposed by Jeong was recreated by using only fully-indicating or minimum cost indicant cover implementations of each function-block. However, the manual process of Jeong was automated using clustering and prime indicant synthesis techniques.
- Full block-level relaxation: where the indication of individual variables within the function block was selected by the relaxation process.

The Block-level results shown in table 1 are for 4 input function blocks only. Simulations were performed for a range of function block sizes for each implementation. Unfortunately, due to space limitations, the results for all cluster sizes can't be reproduced here. However, simulations showed that networks constructed from 4 inputs produced the best results in all three metrics.

All of the block-level procedures demonstrate an improvement in performance and energy over the gate-level relaxation procedure. However, for the prime-indicant synthesis circuits and Jeong's block-level relaxation technique, this comes at the cost of additional area overhead. The new full block-level relaxation technique demonstrates significant reductions in latency (26.4%)and energy (48.0%) over gate-level reduction, while maintaining approximately the same area as the gate-level implementations (4.7% decrease). The increase in granularity of the new block-level relaxation techniques allowed for reductions of 17% in area, 8% in latency and 20% in energy consumption over the block-level relaxation techniques proposed by Jeong.

7. Conclusions

This paper presents a full synthesis procedure to construct indicating combinational datapath circuits from conventional circuit netlists. The procedure constructs function

Benchmark	Gate Level Relaxation			Fully Indicating Synthesis			Jeong Block Level Relaxation			Full Block Level Relaxation		
	Area (10^3 μm^2)	Latency (ns)	Energy (nJ)	Area (10^3 μm^2)	Latency (ns)	Energy (nJ)	Area (10^3 μm^2)	Latency (ns)	Energy (nJ)	Area (10^3 μm^2)	Latency (ns)	Energy (nJ)
c1355	74.2	19.7	244.7	73.3	15.4	161.1	57.7	12.7	104.4	46.93	10.9	85.5
c17	1.21	2.34	3.78	0.89	2.12	1.17	0.89	2.12	1.18	0.89	2.12	1.18
c1908	85.7	18.4	295.1	102	18.6	228.1	96.2	16.7	213.3	79.5	14.4	173.9
c3540	178	26.6	645.3	312	31.3	708.5	258	23.9	578.1	229	18.9	492.6
c432	33.4	19.6	116.4	56.7	16.8	118.5	47.6	15.3	98.6	34.2	13.6	64.8
c499	57.1	14.6	190.8	79.2	12.7	175.6	55.5	10.7	95.7	40.5	9.95	72.1
c5315	294	21.6	1063	501	22.6	1171.9	395	20.5	859.4	311	19.9	638.7
c6288	310	62.3	1112.3	365	35.1	640.3	347	33.9	608.5	319	33.9	550.2
c7552	392	23.8	1541.8	538	21.9	1299.7	442	19.6	1036.0	362	18.0	768.9
c880	51.1	15.6	167.5	99.6	17.5	191.9	79	12.4	148.8	61.4	12.5	109.6
Average Improvement				**-39.6**	**7.44**	**15.6**	**-16.3**	**20.1**	**33.8**	**4.7**	**26.4**	**48.0**

Table 1: Experimental Results of Block-Level Relaxation Synthesis

blocks from the gates in the original netlist and synthesises them using indicating synthesis techniques. Unlike other function-block level techniques, the approach presented in this paper is fully automated. Significant improvements in area, performance and energy over other gate-level and block-level synthesis techniques were demonstrated.

8. References

[1] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Coping with the variability of combinational logic delays", *Proc. ICCD-04*, 2004.

[2] C. Jeong, S. M. Nowick, "Optimization of robust asynchronous circuits by local input completeness relaxation" *Proc. ASPDAC-07*, 2007.

[3] C. Jeong, S. M. Nowick, "Block-Level Relaxation for Timing-Robust Asynchronous Circuits Based on Eager Evaluation", *Proc. ASYNC-08* 2008.

[4] S. P. Khatri, R. K. Brayton, A. Sangiovanni-Vincentelli, "Cross-talk Immune VLSI Design using a Network of PLAs Embedded in a Regular Layout Fabric", *ProcICCAD-00*, 2000.

[5] A. Kondratyev, K. Lwin, "Design of Asynchronous Circuits by Synchronous CAD Tools", *Proc. DAC-02*, 2002.

[6] M. Ligthart, K. Fant, R. Smith, A. Taubin, A. Kondratyev, "Asynchronous Designs using Commercial HDL Synthesis Tools", *Proc. ASYNC-00* 2000.

[7] D. E. Muller, "Asynchronous Logics and Application to Information Processing", *Proc Switching Theory In Space Technology*, 1963.

[8] S. M. Nowick, D. L. Dill, "Exact Two-Level Minimisation of Hazard-Free Logic with Multiple-Input Changes", *IEEE Trans. CAD*, v. 14(8), 1995.

[9] R. L. Rudell. "Logic Synthesis for VLSI Design", *PhD thesis*, University of California at Berkeley, 1989.

[10] J. Sparsø, J. Staunstrup. "Delay Insensitive Multi Ring Structures", *Integration, the VLSI Journal*. v15(13), 1993.

[11] C. Seitz. "System Timing", *Chapter 7 in C.A. Mead and L.A. Conway, editors, Introduction to VLSI systems*, Addison-Wesley, 1980.

[12] W. B. Toms, D. A. Edwards, "Prime Indicants: A Synthesis Method for Indicating Combinational Logic Blocks", *Proc. ASYNC-09*, 2009.

[13] V.I. Varshavsky, ed. "Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems", *Klewer Academic Publishers*,1990.

[14] T. Verhoeff, "Delay-insensitive codes – an overview", *Distributed Computing*, v. 3(1), 1988.

[15] Y. Zhou, D. Sokolov, and A. Yakovlev, "Cost-aware synthesis of asynchronous circuits based on partial acknowledgement. *Proc. ICCAD-06*, 2006.