Prime Indicants: A Synthesis Method for Indicating Combinational Logic Blocks

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 is complex and expensive. This paper presents a novel method for synthesising indicating implementations of arbitrary encoded function blocks. The synthesis method reduces the cost of the implementations by distributing indication between the individual outputs of a function block. Covers are constructed by determining the minimal cost set of Prime Indicants which are required to indicate all of the input transitions of the function block. The results of the procedure are demonstrated on a wide range of combinational logic blocks and show a reduction in literal count of between 38-99%.

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 [2], becomes complex. Self-timed circuits [14], 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 within 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, the datapath must transition into a known (spacer) state in between every transition.

In order to be tolerant to variable delays within circuit components, self-timed circuits use a process called *indication* (or acknowledgement) where the arrival of a set of outputs indicates to the environment that the internal gates of a circuit are in a steady state and the circuit is ready to accept more input. The aim of the work presented in this paper is to minimise the cost of implementing a function block by distributing the indication between all of the outputs of the block. A cover for each function is constructed from the *prime indicants* of the function block; where each prime indicant is either necessary for the function (a *prime implicant*) or indicates an input transition not indicated by any other function. The prime indicants of a function block are highly dependent on the implementations of the individual functions and cannot necessarily be constructed from the prime implicants of each function. The paper presents an efficient method of computing a low cost prime indicant cover for arbitrarily encoded function blocks. The method employs heuristic UCP solvers, with several constraints and novel cost functions.

1.1 Existing Indicating Synthesis Approaches

A popular method of constructing self-timed datapath circuits is using an approach called *desynchronisation* [8][2], where conventional synthesis tools are used to synthesise a gate-level network which is then converted into a self-timed network by expanding each gate into its dual-rail equivalent. This approach allows large self-timed datapaths to be constructed relatively easily. Furthermore, several recent techniques have been developed that significantly reduce the cost (in area, power and delay) of the initial network by using techniques such as *weak-indication* [4][19] and *relative timing* [3].

The use of conventional synthesis tools in the desynchronisation flow has several drawbacks. Firstly, it restricts the self-timed implementations to dual-rail (or in some cases 1-of-4) encoding and so other lower-power or higher-density DI-encodings cannot be employed. Secondly, the behaviour and costs of self-timed networks are significantly different from those of conventional networks and so optimising a datapath for one style may significantly increase the cost of an implementation in the other.

More recently, a block-level approach to synthesis has been proposed [5], where datapaths are constructed from function-blocks connected by encoded channels. As a composition of indicating blocks is itself indicating [14], each block may be specified and synthesised independently without altering the indication of the whole datapath, meaning large self-timed datapaths can be efficiently created. Furthermore, the block-level approach alleviates some of the problems of desynchronisation by allowing much more freedom in the choice of encoding and, as each block has several outputs, can reduce implementation costs by allowing indication to be shared between multiple outputs of the block. In [5], block-level implementations were restricted to dual-rail, and the focus of the work was on selecting the implementation of a block from a restricted set of possible implementations.

1.2 Hazard-Free Combinational Logic Synthesis

A related topic is that of *hazard-free* two-level combinational logic minimisation [11][15]. Unlike indicating logic, hazard-free logic operates under *fundamental mode* assumptions: where the environment uses timing constraints to determine when the circuit has stabilised. In order to be hazard-free, a logic implementation must ensure that no glitches occur on the output of a function during a *multiple input change*: where the inputs transition from one function minterm to another. In hazard-free logic minimisation, multiple input changes from minterm A to minterm B are described by a *transition cube*, $[A,B]$, which contains all of the minterms that may be reached during the MIC. The transition cubes of a specified set of MICs are used to control and constrain the minimisation of function f to construct a hazard free cover, F:

- *Required Cubes* – A transition cube $[A,B]$ where $f(A) = 1$ and $f(B) = 1$ ($[A,B]$ may be all or part of a MIC) is a required cube and must be fully contained within a cube of F.
- *Privileged Cubes – A* transition cube $[A,B]$ where $f(A) = 1$ and $f(B) = 0$ is a privileged cube and must not be intersected by any cube of F that does not contain A.

Several minimisation techniques have been developed based on these constraints that can efficiently synthesise two-level hazard-free implementations.

The indicating logic synthesis techniques described in this paper share similarities with hazard-free synthesis techniques, but the constraints of indicating logic are different to those of hazard-free logic. The allowed-transition sets constrain the MICs within indicating circuits to a small subset of possible transitions and the encoding of data in self-synchronising codes greatly simplifies the complexities of ensuring that implementations are hazard-free. However, the requirement that the output of a function indicates all input transitions make indicating logic synthesis complex. The required cubes of the hazard-free synthesis procedure can be constructed by observing the output value of a function during an MIC, whereas the requirement of a function to indicate input transitions requires the observation of individual inputs during MIC.

The structure of the paper is as follows: section 2 introduces the theory of indication and outlines the requirements of a function block to be indicatable; section 3 introduces a canonical architecture for implementing any indicating function, and outlines the conditions under which this may be optimised; section 4 introduces the concept of a *Prime Indicant* and discusses how a prime indicant cover may be generated; section 5 describes each of the synthesis algorithms in detail; section 6 presents the results on a range of sample function blocks; conclusions and future work are outlined in section 7.

2. Indicating Networks

There are several different models that may be used to explore the principles of indication. In this paper we employ a model developed specifically for combinational logic circuits by Varshavsky [17]. Varshavsky defines indication in terms of multiple input changes called *allowed-transitions sets* that occur within *self-synchronising code-systems*. In this section Varshavsky's basic model is reviewed and the extensions necessary to reason about the indication of individual variables are presented. This augmented model is then used to verify the canonical architecture presented in the section 3.

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 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: $P_1 \times \ldots \times P_n \to P_f$. 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:

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

  where $S \subseteq P_i$. If v_i is a binary variable then $v_i^{\{1\}}$ is written as v_i, $v_i^{\{0\}}$ is written as $\bar{v}_i$ and $v_i^{\{0, 1\}}$ as v_i*. The notation $\tilde{v}_i$ is used to denote either v_i or $\bar{v}_i$ (but not 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 for y.
- An *implicant* of a (*boolean*) function is a product term which contains no minterm of the functions offset.
- 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 the minterms of the on-set and no minterms of the off-set.

2.2 Transition Sets

Transition Sets (TS) occur between the values of a set of binary variables ($V = \{v_1, \ldots, v_n\}$). Each value is described in terms of binary vectors, called *combinations*. Like a multiple input change in hazard-free logic each TS, (*a-b*), also has an associated transition cube [*a*,*b*], which contains all possible combinations reached between *a* and *b*. Furthermore, each TS has two associated products:

- *transition constant term* – the product of the variables which do not change in the TS:

  $$\omega(a, b) = \bigwedge_{j=1}^{n} \alpha_j \quad :$$

  where:

  $$\alpha_j = \begin{cases} \bar{v}_j & if \;\; a_j = b_j = 0 \\ v_j & if \;\; a_j = b_j = 1 \\ 1 & if \;\; a_j \neq b_j \end{cases}$$

- *transition variation term* – the product of the variables that change in the TS:

  $$\varepsilon(a, b) = \bigwedge_{j=1}^{n} \beta_j \quad :$$

  where:

  $$\beta_j = \begin{cases} \bar{v}_j & if \;\; a_j = 1, b_j = 0 \\ v_j & if \;\; a_j = 0, b_j = 1 \\ 1 & if \;\; a_j = b_j \end{cases}$$

There exist several important properties of transition sets and their combinations:

- The *internal transition cube*, (*a*,*b*), of a TS contains all of intermediate combinations between *a* and *b* ($[a, b] - \{a, b\}$).
- A transition set is *regular* if no variable changes value more than once in the transition set (a *monotonic MIC*).
- Combinations *a* and *b* are *adjacent* (w.r.t. variable v_j) if $\varepsilon(a, b) = \tilde{v}_j$. For each TS (*a-b*), the set *N*(*a*,*b*) contains all the combinations adjacent to *b* in [*a*,*b*]. For each $a^i \in N(a, b)$ an a*djacent TS*, (a^i,*b*) may be constructed. The number of elements in *N*(*a*,*b*) is equal to the number of variables in $\varepsilon(a, b)$.
- Two combinations, *a* and *b*, are *comparable* if the variables in the transition variation term $\varepsilon(a, b)$ are either all uncomplemented or all complemented. If *a* and *b* are *incomparable,* $\varepsilon(a, b)$ contains both complemented and uncomplemented variables and there exists a pair of variables, v_i and v_j, who transition in opposing directions during the TS. Therefore, the values of v_i and v_j differ within each combination (i.e. $a_i \neq a_j, b_i \neq b_j$) and between each combination (i.e. $a_i \neq b_i, a_j \neq b_j$).

2.3 Self-Synchronising Code Systems

A code system, *Z*, contains an alphabet of values for a set of variables, V. For each value $a \in Z$, the code system defines a subset of values *Z*(*a*) which may appear after *a*. A TS is called an allowed-transition set (*ATS*) if the following conditions are satisfied:

Definition 2.1 *Transition set (a-b) in code system Z is* ***allowed*** *if the following properties are satsified:*

- $a \in Z, b \in Z(a)$
- $a \neq b$
- (*a-b*) is regular.
- $t \notin Z(a)$ for all intermediate combinations $t \in (a, b)$.

A code system in which all transition sets are allowed is called a *self-synchronising code system* (SSC). In an SSC the completion of an ATS (*a-b*) can be determined by the presence of the value *b* on the code system variables regardless of the order or timing of intermediate transitions.

In *two-phase* SSCs, all ATS occur between a data value, *d*, from the set, *D*, of all valid data values and a spacer, *s*, from the set of all spacer values, *S*, and so $Z = D \cup S$. In this paper only two-phase code systems with a single spacer value ($|S| = 1$) are considered. In order to ensure that all transition sets are allowed, the combinations in *D* are all incomparable with each other but comparable to the spacer value. The set of data values is therefore equivalent to a *DI-code* [18].

In the remainder of the paper, we use (*a-b*) (and also (*k-l*)) to denote an arbitrary (*d-s*) or (*s-d*) ATS. We use (*a*/*b*) to denote either (*a-b*) or (*b-a*).

2.4 Combinational Logic

Combinational logic function blocks consist of *p* variables: A set of *n* inputs, $X = \{x_1 = v_1, \ldots, x_n = v_n\}$, and a set of *m* outputs, $Y = \{y_1 = v_{n+1}, \ldots, y_m = v_p\}$. A TS, (*a-b*) on code system *X*, causes a subsequent TS, (*k-l*), on code system *Y*. The operation of the block is determined by two multi-valued functions:

- $F: P_{D^X} \rightarrow P_{D^Y}$ where

  $$P_{D^X} = \{\alpha_0 = d_0^X, \alpha_1 = d_1^X, \ldots, \alpha_{|D^X|-1} = d_{|D^X|-1}^X\}$$

  $$P_{D^Y} = \{\alpha_0 = d_0^Y, \alpha_1 = d_1^Y, \ldots, \alpha_{|D^Y|-1} = d_{|D^Y|-1}^Y\}$$

  F maps data values $D^X \in X$ to data values $D^Y \in Y$

- $G: P_{S^X} \rightarrow P_{S^Y}$ where

  $$P_{S^X} = \{\alpha_0 = s^X\}$$

  $$P_{S^Y} = \{\alpha_0 = s^Y\}$$

  G maps spacer values $S^X \in X$ to spacer values $S^Y \in Y$.

As only single spacer code systems are described in this paper $G(X^{\{0\}}) = Y^{\{0\}}$ for all function blocks.

Example 2.1 *A* is a combinational logic block with 4 inputs (*n* = 4) and 2 outputs (*m* = 2). The input code system *X* consists of four data values and a single spacer:

$$D^X = \{d_0^X = 1010, d_1^X = 0110, d_2^X = 1001, d_3^X = 0101\}$$

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

Code system X contains eight TS:

$$(s^X - d_0^X), (s^X - d_1^X), (s^X - d_2^X), (s^X - d_3^X)$$

$$(d_0^X - s^X), (d_1^X - s^X), (d_2^X - s^X), (d_3^X - s^X)$$

For transition $(s^X - d_0^X)$:

$$\omega(s^X, d_0^X) = \overline{x_2}\overline{x_4}, \ \varepsilon(s^X, d_0^X) = x_1x_3$$

and

$$N(s^X, d_0^X) = \{1000, 0010\}$$

The output code system, Y, consists of two data values and a single spacer:

$$D^Y = \{d_0^Y = 10, d_1^Y = 01\}$$

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

Code system Y contains four TS:

$$(s^Y - d_0^Y), (s^Y - d_1^Y), (d_0^Y - s^Y), (d_1^Y - s^Y)$$

For transition $(s^Y - d_0^Y)$:

$$\omega(s^Y, d_0^Y) = \overline{y_2} \quad , \quad \varepsilon(s^Y, d_0^Y) = y_1$$

and

$$N(s^Y, d_0^Y) = \{00\}$$

$F(X)$ is defined as follows:

$$Y^{\{0\}} = X^{\{0, 1, 2\}}$$

$$Y^{\{1\}} = X^{\{3\}}$$

Therefore, the TS $(s^X - d_0^X)$, $(s^X - d_1^X)$ and $(s^X - d_2^X)$ on X cause the TS $(s^Y - d_0^Y)$ on Y and the TS $(s^X - d_3^X)$ on X causes the TS $(s^Y - d_1^Y)$ on Y.

Varshavsky defined the requirements for an indicating implementation to be constructed for a function block:

Definition 2.2 *In order for a function block to be* ***indicatable****, the following conditions must be upheld*:

- Code Systems X and Y must be SSC.
- The functions F and G must be completely specified.

Effectively, definition 2.2 means that an indicating implementation can be constructed for any function block with DI-encoded inputs and outputs (functions can be made completely specified by assigning any unspecified input values to arbitrary output values).

To determine the role of individual variables within the ATS of the function block each variable, v_i, has associated input and output code systems $X_i = S^X \cup D^{X_i}$ and $Y_i = S^Y \cup D^{Y_i}$, where:

$$D^{X_i} = \begin{cases} d_j^X \big| \ \varepsilon(s^X / d_j^X) \subseteq \tilde{v}_i & if \quad 1 \le i \le n \\ d_j^X \big| \ \varepsilon(s^Y / d_k^Y) \subseteq \tilde{v}_i, Y^{\{k\}} = F(X^{\{j\}}) & if \quad n+1 \le i \le p \end{cases}$$

$$D^{Y_i} = \left\{ d_k^Y \big| \ Y^{\{k\}} = F(X^{\{j\}}), d_j^X \in D^{X_i} \right.$$

D^{X_i} contains the data values of the ATS in X that involve a transition on v_i. If v_i is an input, D^{X_i} contains the data values from the ATS of X in which v_i transitions. If v_i is an output, D^{X_i} contains those data values in X, which result in ATS (k-l) in Y (as determined by function F), where (k-l) contains a transition on v_i. The set D^{Y_i} contains the data values of Y, that result from applying function F to the data values of D^{X_i}. As they are subsets of SSCs X and Y, code systems X_i and Y_i are also self-synchronising. Multi-valued functions $F_i\colon P_{D^{X_i}} \to P_{D^{Y_i}}$ and $G_i\colon P_{S^{X_i}} \to P_{S^{Y_i}}$ describe the mapping between the data and spacer values of the two code systems associated with each variable.

Example 2.2 For variables v_1 (x_1) and v_5 (y_1) from the function block A in example 2.1, the sets D^{X_i}, D^{Y_i} and corresponding multi-valued function (F_i) are:

$$D^{X_1} = \{d_0^X, d_1^X\}, D^{Y_1} = \{d_0^Y\}, F_1\colon Y_1^{\{0\}} = X_1^{\{0, 1\}}$$

$$D^{X_5} = \{d_0^X, d_1^X, d_2^X\}, D^{Y_1} = \{d_0^Y\}, F_5\colon Y_5^{\{0\}} = X_5^{\{0, 1, 2\}}$$

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* (or *encoded expression* [9]) of variable y_i formed by mapping the multi-valued functions F_i and G_i to the binary encoding of code system X.

2.5 Indication

In order for a function block to be indicating, the transitions of individual input variables must be *indicated* by the output functions of the SIF. To describe the process of indication, Varshavsky introduces the term *translation*:

Definition 2.3 *An input transition is translated to the output of a function if its arrival causes the output to transition.*

In order for a function to be indicating, all input transitions must be *translatable* to the output of a function *in a single step*. The ability is required because not all input transitions in an ATS 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 (the last input transition occurs “a single step” before the output). This property allows the function indicate all input transitions regardless of the order of their arrival.

Varshavsky uses *boolean differences* of functions in order 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:

$$\frac{\partial f_j}{\partial x_i} = f_{jx_i} \oplus f_{j\overline{x_i}}$$

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 2.4 *An output, y_j, translates an input transition on input x_i in a single step in the ATS (a-b), if for each adjacent TS, (a^i-b), where $a^i \in N(a, b)$ and $\varepsilon(a^i, b) = x_i$ or $\varepsilon(a^i, b) = \bar{x}_i$, the values of the variables in $\omega(a^i, b)$ form a solution to the equation:*

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

Example 2.3 For the function block *A* from example 2.1, function y_1:

$$y_1 = x_1x_3 + x_2x_3 + x_1x_4$$

The transition $(s^X - d_0^X)$ has the transition constant and variance terms:

$$\omega(s^X, d_0^X) = \bar{x}_2\bar{x}_4,\ \varepsilon(s^X, d_0^X) = x_1x_3$$

and the adjacent combinations:

$$N(s^X, d_0^X) = \{n^1 = 1000, n^2 = 0010\}$$

Considering the adjacent TS (n^1, d_0^X), the transition on x_3 is translated by y_1 as the value of the variables in the transition constant term form a solution for the equation generated from the boolean difference of the variable in the variation term:

$$\varepsilon(n^1, d_0^X) = x_3 \text{ and } \omega(n^1, d_0^X) = x_1\bar{x}_2\bar{x}_4$$

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

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

The concept of translation is used to define the indication of input transitions by an SIF:

Definition 2.5 *An SIF **indicates** the input transitions of an ATS (a-b) if for each adjacent TS (a^i-b) where $a^i \in N(a, b)$, and $\varepsilon(a^i, b) = x_i$, there exists an output, y_j, which translates the transition on input x_i in a single step.*

In order to reason about the translation of variables in the network the boolean difference equation of 2.4 can be used to determine the following two products for each variable, v_i:

- $\tau_i(a, b)$ is a product of the variables *whose functions translate variable v_i* for the ATS (*a-b*) on input code system *X*.
- $\Upsilon_i(a, b)$ is a product of the variables *translated by function f_i* for the ATS (*a-b*) on input code system *X*.

Example 2.4 For variables v_1 (x_1) and v_5 (y_1) and ATS $(s^X - d_0^X)$ from the function block *A* in example 2.1:

$$\tau_1(s^X, d_0^X) = y_1,\ \Upsilon_1(s^X, d_0^X) = \varnothing$$

$$\tau_5(s^X, d_0^X) = \varnothing,\ \Upsilon_5(s^X, d_0^X) = x_1x_3$$

Using these definitions the condition for an SIF to be indicating can be defined:

Definition 2.6 *In order for an SIF to be indicating every input transition in X is translated to the outputs Y in one step for all ATS:*

$$|\tau_i(a, b)| \geq 1 \quad \Big| \quad (a - b) \in X^i \ \ for \ \ 1 \leq i \leq n$$

3. Canonical Architecture

The principles of indication can be used to verify a canonical architecture for indicating SIFs in which any function block adhering to definition 2.2 may be implemented. As described in section 2.4, each cover function, f_i, in an SIF is the encoded function generated by mapping the multi-valued functions F_i and G_i to the binary encoding of code system *X*. Therefore, each output function, f_i, is constructed from two separate functions: f_i^1 – the encoding function of F_i (which covers spacer to data ATS) and f_i^2 – the encoding function of G_i (which covers data to spacer ATS). The two functions are composed sequentially:

$$f_i(X) = f_i^1(X) + (f_i(X) \cdot \overline{f_i^2(X)})$$

The minterms of the encoded functions are cubes that correspond to the combinations in D^{X_i} or S^X. In the canonical architecture however, encoded function $f_i^1(X)$ is constructed from the sum of the *transition variation terms* of all of the spacer to data ATS in code system X^i:

$$f_i^1(X) = \varepsilon(s^X, d_1^X) + \ldots + \varepsilon(s^X, d_n^X) \Big| \ d_j^X \in D^{X_i}$$

and $f_i^2(X)$ is a single cube corresponding to the spacer value.

$$f_i^2(X) = s^X$$

Theorem 3.1 *If f_i is a cover function of SIF A' implemented in the canonical architecture, then f_i translates all input transitions of ATS in the code system X^i.*

Proof If f_i contains a single cube, *c*, then the cofactors of any literal, x_j, in *c* will be: $f_{ix_j} = c/x_j$ and $f_{i\bar{x}_j} = \varnothing$ where c/x_j is a cube containing all the literals of *c* except x_j. Therefore, *f* translates all of the transitions of literals within *c* for any ATS in X^c.

Definition 2.1 states that no combination (other than *a* or *b*) within the transition cube of ATS (*a-b*) can be a member of a code system. Therefore, the combinations of D^X are *mutually-exclusive* within the ATS of code system *X*. As all data values within a self-synchronising code system are incomparable, there exists a pair of literals which differ within and between each data value and hence the transition variation terms of each ATS are also *mutually-exclusive* within *X*. This means each transition variation term can be used to distinguish the combinations of each data value and, as only the variables with the transition variation term need to be indicated, there is no need to include the variables in the transition constant term within the associated cube of the function. As each transition variation term is mutually-exclusive, for each ATS $(s^X - d_j^X)$ in code system X^i, only one cube of $f_i^1(X)$, *c*, (corresponding to $\varepsilon(s^X, d^X{}_j)$) evaluates to true ($c = 1$). In this case, the boolean difference equation behaves the same as the single cube case and $f_i^1(X)$ indicates all input transitions in the spacer to data ATS of X^i.

Function $f_i^2(X)$ is a single cube and so it translates all of the input transitions of the data to spacer ATS in X^i.

In single spacer systems the functions $f_i^1(X)$ and $f_i^2(X)$ must be mutually-exclusive and so function $f_i(X)$ translates all transitions of both functions and hence all input transitions in the ATS of X^i. □

If a function is to be implemented as a set of discrete gates (such as an And-Or Sum-Of-Products implementation), the outputs of any intermediate gates become additional variables in the SIF and also need to be translated. In the canonical architecture outlined above, the cubes of $f_i^1(X)$ only translate inputs transitions from the set X^i and so all of the transitions of each $c \in f_i^1(X)$ are translated by f_i. As the function $f_i^2(X)$ contains a single cube corresponding to the spacer combination, it will translate all data to spacer ATS including those not in X^i. Therefore, not all of the transitions of $f_i^2(X)$ are translated and it cannot be implemented as a discrete gate.

The canonical architecture can be implemented in distributed form, where each cube, c, of $f_i(X)$, is constructed from a sequential composition of cubes:

$$c = c_{on} + (c \cdot \overline{c_{off}})$$

where $c_{on} = \varepsilon(s^X, d_j^X)$ and $c_{off} = \varepsilon(d_j^X, s^X)$:

Return-to-Zero (RTZ) SSCs are a class of code systems whose spacer is the all-zero input vector. In RTZ codes, each $\varepsilon(d_i, s)$ contains the negation of all the variables in $\varepsilon(s, d_i)$ and in such systems, c can be implemented in a single C-element as depicted in figure 1. This distributed architecture forms the basis of Muller's Combinational Logic Blocks [10], Sparsø's DIMS architecture [13] and Theseus's NCL-D [8] and has been shown [16] to be equivalent to the Kondratyev's basic gate speed-independent architecture [7].

The requirements of definition 2.2 ensure that if each output function of an SIF translates all of its input transitions the SIF is indicating:

Theorem 3.2 *If A' is an SIF, implementing an indicatable function block A, whose functions are all implemented in the canonical architecture, then A' indicates all transitions of input code system X.*

Proof As the multi-valued functions F and G for function block A are completely specified, for each ATS, (s^X / d_i^X) ,on code system X, there must be corresponding ATS on code system Y, (s^Y / d_j^Y), where $F(X^{\{i\}}) = X^{\{j\}}$ or $G(X^{\{0\}}) = Y^{\{0\}}$. As A' is implemented in the canonical architecture each output variable in $\varepsilon(s^Y / d_j^Y)$ must translate all of the input transitions in $\varepsilon(s^X / d_i^X)$. Therefore, for each $\varepsilon(s^X / d_i^X) \subseteq x_k$:

$$\tau_k(s^X / d_i^X) = \varepsilon(s^Y / d_j^Y)$$

The requirements of an ATS (definition 2.1) mean that $|\varepsilon(s^Y / d_j^Y)| \geq 1$ and hence the SIF is indicating according to definition 2.6. □

The cost of implementing functions in the canonical architecture can be significant in both area and delay. As each output

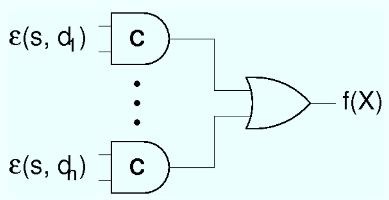

Figure 1: Distributed Canonical Architecture

must translate all input transitions, C-elements are large and no outputs can be produced until all the inputs have transitioned so the SIFs have worst-case latency. In function blocks that have several outputs, there is often more than one output that transitions as a result of each input ATS ($|\varepsilon(s^Y / d_j^Y)| > 1$). In such function blocks it is possible to reduce the cost of implementing each function by distributing the indication of individual input transitions between the outputs of $\varepsilon(s^Y / d_j^Y)$ without violating the indication of the SIF. This is achieved by reducing the number of literals in the cubes of the output functions. Once the literals of a cube, c, have been reduced, c_{on} and c_{off} are no longer complete transition variation terms and therefore the theorems 3.1 and 3.2 no longer hold. To ensure a SIF remains indicating during the minimisation process, we enforce the following restrictions:

Definition 3.1 *The on-sets* $C_{on} = \{c_{on}^1, \ldots, c_{on}^n\}$ *and off-sets* $C_{off} = \{c_{off}^1, \ldots, c_{off}^n\}$ *of each function, f_j, must have the following properties:*

i. Each c_{on}^i contains one or more $\varepsilon(s^X, d_k^X) \in X^j$ and the associated code system, $X_i^{on} = S^X \cup DX_i^{on}$, is constructed from:

$$DX_i^{on} = \left\{ d_k^X \middle| \ \varepsilon(s^X, d_k^X) \subseteq c_{on}^i \right.$$

ii. Each c_{off}^i contains one or more $\varepsilon(d_k^X, s^X) \in X^j$ and the associated code system $X_i^{off} = S^X \cup DX_i^{off}$, is constructed from:

$$DX_i^{off} = \left\{ d_k^X \middle| \ \varepsilon(d_j^X, s^X) \subseteq c_{off}^i \right.$$

iii. $X_i^{off} \subseteq X_i^{on}$

iv. The cubes of C_{on} must be mutually-exclusive

v. $X^j = \bigcup_{i=1}^{n} X_i^{on}$

Any function, f_j, upholding these definitions will translate the input transitions on the literals within each of its cubes for all the ATS in X^j:

Theorem 3.3 *An output cover function, f_j, which upholds the restrictions of definition 3.1 has the following properties:*

$$\Upsilon_j(s^X, d_k^X) = c_{on}^i \Big| \ c_{on}^i \in C_{on}, (s^X - d_k^X) \in X_i^{on}$$

and

$$\Upsilon_j(d_k^X, s^X) = c_{off}^i \Big| \ c_{off}^i \in C_{off}, (d_k^X - s^X) \in X_i^{off}$$

Proof As each cube $c_{on}^i \in C_{on}$ is mutually exclusive, the results of theorem 3.1 apply and each c_{on}^i translates all of its literals for ATS in X_i^{on}. As each c_{off}^i is sequentially composed with the corresponding c^i_{on}, it can only translate input transitions for data to spacer transitions where c^i_{on} is initially true:

$$\left\{ (d^X_k - s^X) \middle| \ d^X_k \in DX^{on}_i \right.$$

As each c^i_{on} is mutually-exclusive, each $(c_i \cdot \overline{c^i_{off}})$ is also mutually exclusive and hence c^i_{off} translates all of its literals for ATS in X^{off}_i. □
This result can be used determine the conditions under which a minimised SIF is indicating:

Theorem 3.4 *If A' is an SIF constructed from functions upholding definition 3.1, then A' is indicating iff for each input transition in each ATS, $\tilde{x}_i \subseteq \varepsilon(s^X / d^X_m)$, there exists some function, f_j with cube c^l_{on}* or *c^l_{off} where:*

$$\varepsilon(s^X, d^X_m) \subseteq c^l_{on} \subseteq \tilde{x}_i \text{ or } \varepsilon(d^X_m, s^X) \subseteq c^l_{off} \subseteq \tilde{x}_i$$

Proof This follows directly from theorem 3.3 and definition 3.1. □

Theorem 3.4 can be used to create the following definition that ensures the minimisation procedure maintains the indication of the original SIF.

Definition 3.2 *For each ATS (a-b) of code system X the product of all the (on-set or off-set) cubes c^j_* which contain the transition variation term $\varepsilon(a, b)$ translates all of the variables in the original transition variation term if:*

$$\wedge c^j_* = \varepsilon(a, b) | \ \ \varepsilon(a, b) \subseteq c^j_*$$

A minimisation procedure can now be constructed for a set of functions whose cubes are the transition variation terms of the ATS of code system *X*. By implementing functions using the restrictions of definition 3.1 and ensuring definition 3.2 is upheld throughout minimisation procedure, an indicating SIF implementation is ensured. Furthermore as definition 3.2 applies to both spacer to data and data to spacer transitions, the minimisation procedure can be used to minimise both the on-set and off-set of each function.

4. Prime Indicants

The aim of minimising an indicating SIF is to reduce the logical complexity of the output functions by exploiting ATS where more than one output translates an input transition ($|\tau_i(a, b)| > 1$). For each ATS of this type, literal x_i can be removed from the corresponding cube of the functions of all but one of the variables in $\tau_i(a, b)$ and the indication of the SIF is maintained (providing the resulting functions uphold the restrictions of definition 3.1). The minimisation procedure must determine which literals to remove from which functions in order to minimise the total cost, in literal counts, of the SIF. In conventional logic minimisation, a minimum cost implementation of a function can be constructed from a cover of the *prime implicants* of a function [12]. In indicating logic, the cubes of a function serve two purposes: to implement each function correctly *and* to indicate transitions on the inputs. The restrictions of definition 3.1 mean a minimum cost implementation of a function cannot necessarily be constructed solely from a covering of its prime implicants. Furthermore, any optimisation from the canonical architecture is dependent on the implementation of other functions in the SIF. Therefore, the concept of *prime indicants* is introduced:

- An *indicant* of a SIF is a cube, *c*, of a function, f_j, which translates the transitions of all of its literals for all ATS in the code system X^c, where $X^c \subseteq X^j$.
- A *prime indicant* of a SIF is an cube, *c*, of f_j, such that *c* is an indicant of the SIF and is either:
  - *i.* a *prime implicant* of f_j.
  - *ii.* indicates transitions in the SIF not covered by prime indicants of other functions in the SIF (an *essential prime indicant*).

Theorem 4.1 *A minimum cost indicating SIF can be created from a covering of prime indicants.*

Proof A SIF is indicating if, and only if, the output functions translate all the transitions of the inputs of the SIF as well as the transitions of the cubes of each function. If a single cube is translated by an output function, all of its input transitions are also translated. Therefore, *any indicated cube must be an indicant* and an indicating SIF is constructed solely from indicants. In order to maintain indication, we can only remove literal x_i from a cube that translates it in ATS $(a - b)$ if $|\tau_i(a, b)| > 1$. Consider a transition on x_i during $(a - b)$, where the x_i is translated by two cubes c_j and c_k. If c_j and c_k are prime implicants then neither cube can be expanded further without intersecting the off-set of their respective functions. If c_j and c_k are not prime implicants then one cube, c_j could be expanded by removing x_i and cube c_k will then become an essential prime indicant. Therefore any indicating SIF implementation may be reduced by expanding non essential indicants into prime implicants. □

As the expansion of each cube can affect the expansion of other cubes, there are many possible prime indicant covers for a function block. Constructing a minimum cost prime-indicant cover by expanding individual cubes of the canonical architecture is prohibitively expensive for all but the simplest function blocks. A better approach is to construct an initial SIF from a prime implicant cover of each function and *reduce* the function cubes by adding literals until they are all prime indicants. All of the procedures presented in this paper are based on identifying a *reduction set*: a subset of prime implicants to reduce that will incur the smallest additional cost to the cover. The cost of generating an indicating SIF by this method depends on the initial prime implicant cover. The minimum cost prime indicant cover generated from an arbitrary prime implicant cover may not be the minimum cost indicating SIF of the function block. However, performing the synthesis routine on all possible prime implicant covers of each function is infeasible and so following assumption is used:

Assumption 4.1 *A low cost non-prime implicant cover can be generated by applying a minimum cost reduction set to a mini-*

mum cost prime-implicant cover.

A synthesis procedure to find a low cost indicating SIF for an arbitrary encoded function block can then be constructed from a series of reduction set selection procedures of an initial prime implicant cover of the output functions of the function block. Initially the prime implicant cover of each function must be reduced to ensure it upholds the requirements of definition 3.1 and all the cubes are indicants. Then, the individual indicant covers are transformed into an indicating SIF by determining input transitions that are not translated and applying a reduction set to translate them. Finally, the off-sets of functions can be further reduced as they have fewer restrictions than the on-sets.

The problem of generating reduction sets for all of the synthesis procedures is equivalent to the *unate covering problem* (UCP) of finding a minimum column set covering a matrix. The procedures described in this paper are implemented using a heuristic UCP solver (which approximates a minimum cost cover), based on the MINCOV [12] solver used in the Espresso PLA minimisation package [1]. The cost functions of the UCP solvers use a simple metric based on literal counts in order to maximise the number of don't care values that may be exploited by multi-level optimisation. It is possible to use more complex cost functions (such as those used in the UCP-based desynchronisation optimisation techniques [4][5][19]) if physical implementations are targeted.

A heuristic UCP solver was chosen as this forms the basis of many existing combinational logic synthesis algorithms. However, the covering problems presented in this paper are more complex than those in conventional synthesis and so each reduction set selection problem may be better implemented using SAT solvers or linear programming techniques. Investigation into the efficacy of using such methods is the subject of future work.

5. Prime Indicant Generation

5.1 Indicant Cover

The first stage of the synthesis process is to construct a minimum cost indicant cover for each function. Definition 3.1 outlines the minimum requirements for each cube of a function to be an indicant. The impact of these requirements on the synthesis process is as follows:

- Definition 3.1.*i* states that each cube must contain one or more of the transition variation terms $\varepsilon(a, b) \in X^j$. Therefore, the transition variation terms become the *required cubes* of the minimisation procedure: each term must be completely covered by a prime implicant in the final cover.
- Definition 3.1.*iii* states that the code system of each off-set cube X_i^{off} is equal to or a subset of the code system of the associated on-set cube, X_i^{on}. A minimum cost indicating cover can therefore be constructed by considering the on-set functions separately and optimising the off-set later (see section 5.3).
- Definition 3.1.*iv* states that each cube of the on-set function must be mutually-exclusive within code system X.

Therefore, an indicant cover is constructed by generating a mutually-exclusive covering of the prime implicants of the on-set transition variation terms of each function.

In order to generate the prime implicants of a function a strategy similar to the expansion process of the EXPAND algorithm in Espresso [1] is used. The EXPAND algorithm expands the cubes of a function individually using two matrices to guide the process. A *blocking matrix* is used to ensure the expanded cube does not intersect with the off-set of the function and the *covering matrix* determines which cubes of the function can be covered by the expansion process. The algorithm proceeds by selecting literals of the current cube to "raise" (become don't care), based on how many other cubes of the function it covers. A cube is fully expanded (and is a prime implicant) when there are no more literals that may be raised without the cube intersecting the off-set. The procedure then removes all the other cubes covered by the expanded cube from the covering matrix before expanding the next cube.

The I-EXPAND procedure follows a similar approach to expanding the cubes of function f_j. The blocking matrix is constructed from the complete combinations of the spacer value, s^X, and all of the data values $d_k \notin D^{X_j}$. The covering matrix is constructed from the transition variation terms of the data values $d_k \in D^{X_j}$. This allows implicants to be expanded into the redundant boolean space within the code systems, as well as between the spacer and data values. A mutually-exclusive cover can be generated in two ways. If the function is very large, the cubes of a function can be expanded sequentially and then added to the blocking matrix of subsequent cubes to ensure that they will remain mutually exclusive. A better approach, provided the function is not too large, is to expand each cube to cover as many cubes as possible and then reduce a subset of the cubes to produce a mutually-exclusive cover.

The reduction set is created from the set of expanded cubes that are not mutually-exclusive. For each cube, c, in this set, all possible implicants of f_j that may be generated by reducing c by literals in its raised set are enumerated. If the expansion process has raised n literals, there are 2^n possible implicants that may be enumerated, and so if n is large this approach may become impractical.

The aim of the covering process is to determine the *minimum independent set of implicants that covers all the required cubes of* f_j. Two implicants are independent if they do not cover the same required cubes. Unlike conventional UCP solving, where are *maximal* independent set of minterms is used to determine a lower bound for the solution, the independent set *forms the solution*. As the required cubes of each function are mutually exclusive, an *independent set* of implicants can always be constructed and the covering process determines the minimum cost solution.

An approximation to a minimum independent set can be generated using a UCP solver. In order to ensure all columns are independent, if a column, *col*, is selected as part of the solution, all columns that intersect each row covered by *col* must be removed from the matrix. This means it is possible that the inclusion of a column into a solution may prevent a full cover being generated. Therefore, before selecting each column *col*, the rows or the matrix *not* covered by *col* must be checked to make sure that they are not covered by a subset of the columns that intersect with *col*. Once a covering has been produced the original prime implicants are removed from the function covers and replaced with the mutually-exclusive implicants.

Example 5.1 Function block A has 6 inputs:

$$X = \{a0, a1, b0, b1, c0, c1\}$$

with alphabet:

$$D^X = \{101010, 101001, 100110, 100101$$
$$011010, 011001, 010110, 010101\}$$

$$S^X = \{000000\}$$

The on-set of function f_j is defined by the transition variation terms:

$$f_j = a0b0c0 + a0b0c1 + a0b1c0 + a1b0c0$$

and the off-set constructed for the blocking matrix is:

$$OFF(f_j) = \overline{a0}\,\overline{a1}\,\overline{b0}\,\overline{b1}\,\overline{c0}\,\overline{c1} + \overline{a0}a1\overline{b0}b1c0\overline{c1} +$$
$$\overline{a0}a1b0\overline{b1}\,\overline{c0}c1 + a0\overline{a1}\,\overline{b0}b1\overline{c0}c1 + \overline{a0}a1\overline{b0}b1\overline{c0}c1$$

The expanded cubes of f_j (with their raised set in brackets) are:

$$EXP = \{a0b0 \quad (c0, c1), a0c0 \quad (b0, b1), b0c0 \quad (a0, a1)\}$$

The total set of implicants is:

$$IMP = \{a0b0, a0b0c0, a0b0c1, b0c0$$
$$a1b0c0, a0c0, a0b1c0\}$$

Figure 2 shows a minimum independent set covering of the all the implicants which results in the function:

$$f_j = a0b0 + a1b0c1 + a0b1c1$$

5.2 Untranslated Inputs

While a minimum cost indicating cover for function, f_j, translates all of the transitions of the input literals in its cubes, it no longer translates the transitions of *all* input variables in the ATS of X^j. Therefore, the properties of the canonical architecture (theorem 3.2) no longer hold and the resultant SIF may not be indicating. The SIF is made indicating by determining the input transitions that are not translated by the indicants that cover each transition variation term (Untranslated input transitions are those that do not appear in any of the covering indicants). A subset of the indicants can then be reduced to translate them. As the indicants of each function are mutually-exclusive they cannot cover the same transition variation terms and the reduction set must be determined from the indicants of all functions in the SIF. Furthermore, each reduced indicant must cover the same terms as the original indicant, otherwise its function will not be covered correctly (definition 3.1.*v*). This means that when reducing an indicant, it must be partitioned into a set of indicants that cover the same terms of the original function and are mutually-exclusive. In some cases, particularly in dual-rail implementations, an indicant reduced by a single literal may be partitioned into two mutually exclusive indicants. However, for some indicants there may not be a mutually-exclusive partitioning other than the set

	a0b0	b0c0	a0c0	a0b0c0	a0b0c1	a0b1c0	a1b0c0
a0b0c0	1	1	1	1			
a0b1c0			1		1		
a0b0c1	1					1	
a1b0c0		1					1

Figure 2: Minimum Independent Set Covering for f_j

Example 5.2 Indicant $I = a0b0$ covers the transition variation terms:

$$\{a0b0c0d0, a0b0c0d1, a0b0c1d0, a0b0c1d1\}$$

Reducing *c* by literal *c0* partitions it in to two indicants:

$$I1 = a0b0c0 \text{ and } I2 = a0b0c1$$

which cover the terms:

$$\{a0b0c0d0, a0b0c0d1\} \text{ and } \{a0b0c1d0, a0b0c1d1\}$$

respectively, and hence are mutually exclusive.

Example 5.3 Indicant $J = a0b0$ (from a different function block) covers the terms:

$$\{a0b0c0, a0b0c1, a0b0c2, a0b0c3\}$$

Reducing *J* by literal *c0* partitions it into four indicants:

$$J1 = a0b0c0, J2 = a0b0c1, J3 = a0b0c2, J4 = a0b0c3$$

A unate recursive algorithm [1] is used to reduce indicant, *I* by literal *x* and partition it into a mutually-exclusive set of indicants. The cofactors I_x and $I_{\bar{x}}$ are created and the terms they cover are calculated. If there is one or more literals that divides the terms covered by $I_{\bar{x}}$ evenly, then a mutually-exclusive partition can be created. Otherwise another variable must be chosen to reduce *both* cofactors and the procedure recurses. The choice of the next reduction variable can have a big impact on the number of indicants a literal is partitioned into:

Example 5.4 Indicant $I = a0$ and covers terms

$$\{a0b0c0, a0b0c1, a0b0c2, a0b0c3$$
$$a0b1c0, a0b1c1, a0b1c2, a0b0c3\}$$

if $c0$ is selected as a reduction variable would result in the partitioning:

$$I1 = a0c0, I2 = a0c1, I3 = a0c2, I4 = a0c3$$

But selecting $b0$ as a reduction variable would result in the partition:

$$I1 = a0b0, I2 = a0b1$$

The reduction variable is selected by choosing the variable that appears in most terms in an effort to reduce the number of indicants produced.

A partitioning set is created for each indicant that covers an untranslated literal. A covering of the partitioning sets is then selected that will minimise the additional cost to the network. In order to minimise the cost to the network, the reductions need to be distributed between the indicants of the network, because when a pair of reductions are applied to an indicant, the number of resultant indicants is multiplied.

Example 5.5 If $I = a0$ and covers the terms:

$$\{a0b0c0, a0b0c1, a0b1c0, a0b1c1\}$$

a0	a1	b0	b1	c0	c1	d0	d1	r0	r1	s0	s1
1	-	1	-	1	-	1	-	1	0	1	0
1	-	1	-	1	-	-	1	1	0	0	1
1	-	1	-	-	1	1	-	1	0	1	0
1	-	1	-	-	1	-	1	1	0	0	1
1	-	-	1	1	-	1	-	1	0	1	0
1	-	-	1	1	-	-	1	1	0	0	1
1	-	-	1	-	1	1	-	1	0	1	0
1	-	-	1	-	1	-	1	1	0	0	1

a0	a1	b0	b1	c0	c1	d0	d1	r0	r1	s0	s1
-	1	1	-	1	-	1	-	0	1	1	0
-	1	1	-	1	-	-	1	0	1	0	1
-	1	1	-	-	1	1	-	0	1	1	0
-	1	1	-	-	1	-	1	0	1	0	1
-	1	-	1	1	-	1	-	0	1	1	0
-	1	-	1	1	-	-	1	0	1	0	1
-	1	-	1	-	1	1	-	0	1	1	0
-	1	-	1	-	1	-	1	0	1	0	1

Figure 3: Truth Table for example 5.6

Reducing *I* by *b*0 results in the partition:

$$I1 = a0b0 \text{ and } I2 = a0b1$$

and reducing *I* by *c*0 results in the partition:

$$I1 = a0c0 \text{ and } I2 = a0c1$$

However applying both reductions together results in the partitioning:

$$\{a0b0c0, a0b0c1, a0b1c0, a0b1c1\}$$

The reduction set selection therefore has to take into account the relationship between reductions of the same indicant.

A reduction set is selected by covering a matrix constructed from the untranslated literals and the indicants that could potentially cover them. The untranslated literals form the rows of the matrix, and the partitioning sets form the columns. Each partitioning set will translate at least two literals (e.g. the partitioning set in example 5.2 translates both *c*0 and *c*1), and therefore any duplicate partitioning sets are removed. The columns constructed by the partitioning sets of each indicant are then grouped into *buckets*. The weight of each column, *j*, is a function of the number of rows covered by a column and the cost of implementing it. The cost of *j* is a function of how many indicants are in the partitioning set associated with *j* (p_j), and how many previously selected columns of the solution, *S*, are in the same bucket as *j*, B_j:

$$C_j = p_j\left(\prod_{i \in B_j \cap S} p_i\right) \tag{4.1}$$

The covering problem therefore becomes more difficult, because the weights of individual columns are no longer independent, and the cost of the existing columns in a covering can change depending on which further columns are selected. Techniques to reduce the size of a matrix such as row and column dominance or Gimpel's reduction [12] can no longer be employed because, at the time the reduction is executed, the final cost of each column can not be determined, making the covering problem larger. However, the solution does have the advantage of distributing the reductions across the indicants.

Example 5.6 A function block contains 6 inputs, and 4 outputs:

$$X = \{a0, a1, b0, b1, c0, c1\}$$

$$Y = \{r0, r1, s0, s1\}$$

The transition variation terms of each function are shown in the truth table of figure 3.

A mutually-exclusive expanded cube covering for the functions is:

Bucket		a0		a1		d0		d1		Bucket		a0		a1		d0		d1	
Partition		b0 b1	c0 c1	b0 b1	c0 c1	b0 b1	c0 c1	b0 b1	c0 c1	Partition		b0 b1	c0 c1	b0 b1	c0 c1	b0 b1	c0 c1	b0 b1	c0 c1
a0b0 c0d0	b0	**1**				1				a0b0 c0d0	b0			**1**		1			
	c0		1				**1**				c0				1		**1**		
a0b0 c0d1	b0	**1**						1		a0b0 c0d1	b0			**1**				1	
	c0		1						**1**		c0				1				**1**
a0b0 c1d0	b0	**1**				1				a0b0 c1d0	b0			**1**		1			
	c1		1				**1**				c1				1		**1**		
a0b0 c1d1	b0	**1**						1		a0b0 c1d1	b0			**1**				1	
	c1		1						**1**		c1				1				**1**
a0b1 c0d0	b1	**1**				1				a0b1 c0d0	b1			**1**		1			
	c0		1				**1**				c0				1		**1**		
a0b1 c0d1	b1	**1**						1		a0b1 c0d1	b1			**1**				1	
	c0		1						**1**		c0				1				**1**
a0b1 c1d0	b1	**1**				1				a0b1 c1d0	b1			**1**		1			
	c1		1				**1**				c1				1		**1**		
a0b1 c1d1	b1	**1**						1		a0b1 c1d1	b1			**1**				1	
	c1		1						**1**		c1				1				**1**

Figure 4: Untranslated Literal Matrix for example 5.6

$$r0 = a0\,,\; r1 = a1\,,\; s0 = d0\,,\; s1 = d0$$

Therefore, transitions on variables *b*0, *b*1, *c*0 and *c*1 are not translated in any term.

The partitioning sets of the four indicants are:

$$a0\text{: } b0/b1 = \{a0b0, a0b1\}\,,\; c0/c1 = \{a0c0, a0c1\}$$

$$a1\text{: } b0/b1 = \{a1b0, a1b1\}\,,\; c0/c1 = \{a1c0, a1c1\}$$

$$d0\text{: } b0/b1 = \{b0d0, b1d0\}\,,\; c0/c1 = \{c0d0, c1d0\}$$

$$d1\text{: } b0/b1 = \{b0d1, b1d1\}\,,\; c0/c1 = \{c0d1, c1d1\}$$

The untranslated literal matrix is shown in figure 4. Each bucket has two columns that correspond to the partitioning sets, and each transition variation term has two rows that correspond to each untranslated literal. It is possible to construct a covering for this matrix using columns from two of the four buckets, for example *a*0 and *a*1, resulting in the implementation with a literal count of 26:

$$r0 = a0b0c0 + a0b0c1 + a0b1c0 + a0b1c1$$

$$r1 = a1b0c0 + a1b0c1 + a1b1c0 + a1b1c1$$

$$s0 = d0\,,\; s1 = d1$$

However, because the weights of all the columns are equivalent in this example, the cost function (4.1), makes selecting columns from the same bucket expensive. The solution highlighted in figure 4, gives a literal count of 16:

$$r0 = a0b0 + a0b1\,, r1 = a1b0 + a1b1$$

$$s0 = c0d0 + c1d0\,,\; s1 = c0d1 + c1d1$$

5.3 Off-set Optimisation

In the distributed canonical architecture, the off-set cube of each indicant is formed from the transition variation term $\varepsilon(d_k^X, s^X)$. In single spacer code systems, data to spacer transition variation terms are not mutually-exclusive and so each off-set cube must be composed sequentially with the corresponding on-set cube ($\varepsilon(s^X, d_k^X)$) *in a single gate*. Because of this, the off-set expansion cannot be completely separated from the on-set expansion, as in the binate covering approach of [19] and must be performed after the on-set minimisation.

Definition 3.1.*iii* states that the code system X^{off} of off set cube, c_{off}, is equal to or a subset of the code system X^{on} of the associated on-set cube, c_{on}. Therefore, off-set of each indicant, *c*, can only translate transitions from the set:

$$\left\{ \varepsilon(d_k^X, s^X) \middle| \; d_k^X \in X_i^{on} \right.$$

The set of inputs translated by the off-set $\Upsilon_c^{off}(d_k^X, s^X)$ does not necessarily have to be the same those translated by the on-set:

$$\Upsilon_c^{off}(d_k^X, s^X) \neq \Upsilon_c^{on}(s^X, d_k^X)$$

It must however fully contain some transition variation term(s) $\varepsilon(d_k^X, s^X)$ where $d_k^X \in DX_{on}^I$.

Example 5.7 The on-set of indicant, *c*,is $c^{on} = a0b0$ and covers two terms of the SIF:

$$\{a0b0c0, a0b0c1\}$$

the off-set, c^{off}, can translate any of the literals in c^{on} or either of the literals $c0$ or $c1$. It cannot however, translate both literals as the product $\overline{c0}\,\overline{c1}$ does not contain any transition variation term of the SIF.

Indicants with a minimised off-set can be implemented in two-level implementations using asymmetric C-elements. If the off-set does not translate any transitions an indicant can be implemented using an AND gate.

There are three strategies for minimising the off-set:

i. *General Minimisation*: the off-sets of each indicant, *c*, can translate a subset of input transitions from the set:

$$\varepsilon(d_k^X, s^X) \in X^{on}$$

ii. *Subset Minimisation*: the off-sets of each indicant, *c*, can only translate an input transition translated by the on-set:

$$\Upsilon_c^{off}(d_k^X, s^X) \subseteq \Upsilon_c^{on}(s^X, d_k^X)$$

iii. *AND Minimisation*: the off-sets can either translate all of the input transitions of the on-set, or none

$$\Upsilon_c^{off}(d_k^X, s^X) = \Upsilon_c^{on}(s^X, d_k^X) \text{ or } \Upsilon_c^{off}(d_k^X, s^X) = \varnothing$$

The resulting SIF can be implemented using only symmetric C-elements and AND gates.

All three strategies are solved using a unate covering of a matrix with the literals of each term as rows and the literals of each indicant as columns. The three strategies differ in the number of columns per indicant, *c*:

i. General Minimisation: one column for each literal of c_{on} and its raised set.

ii. Subset Minimisation: one column for each literal of c_{on} only.

iii. AND Minimisation: one column for each *c*.

In the general minimisation strategy, when a column of *c* is selected, the other columns of *c* are checked and possibly removed from the matrix to ensure each off-set fully contains some transition variation term.

As the cost of implementing additional inputs in asymmetric C-elements is linear, each strategy may be solved using a conventional unate covering solution. However, the results presented in section 6 employ the UCP costing outlined in section 5.2 since this is a good way of distributing the off-set amongst all of the indicants of the network.

6. Results

The aim of the synthesis process described in this paper is to minimise the logical complexity of an indicating SIF, as measured by the combined literal count of all of the functions. The procedures are not targeted at optimal two-level implementations as there is no attempt to share terms between functions. If two-level implementations were desired, it would be possible to maximise the sharing between functions by using multiple output minimisation to determine the initial indicant cover of the SIF. However, the basis of the optimisation procedure is to exploit the differences between function implementations to distribute indication between outputs and so term sharing can reduce the effectiveness of optimisation. Furthermore, the nature of the canonical architecture means that each cube must be implemented within a single-gate. Decomposing indicating SIFs is very difficult, as each internal gate must be indicated by the outputs, which, in general, requires a high degree of sharing between functions. Procedures have already been developed [16] to decompose any SIF (adhering to definition 2.2) so it may be implemented using 2-input gates. However, the cost of such implementations is high. The procedures presented in this paper were developed to construct indicating specifications of function blocks which minimise the sharing between functions and therefore simplify the decomposition process.

Table 1 presents the results of the synthesis algorithms over a range of function blocks. The table shows the total literal count of implementations in the unoptimised canonical architecture, the (best) total literal counts of the optimised implementations and the percentage decrease in literal count. The remaining columns present the optimised on-set literal counts and the optimised off-set literal counts for each of the three schemes outlined in section 5.3. In order to demonstrate the effectiveness of off-set optimisation, the number of C-elements (indicants with one or more literals in the off-set) and AND gates (indicants with no literals in the off-set) for a two-level implementation are given.

The first three examples are *m*-of-*n* adder circuits demonstrating the ability of the synthesis procedure to synthesise blocks with any encoding. The remainder of the circuits are dual-rail encoded blocks from the ISCAS benchmarks. In order to generate suitably sized blocks from the gate level netlists of the ISCAS benchmarks, a greedy clustering algorithm, such as the one described in [6], was implemented. The algorithm traverses the netlist adding gates into clusters until a maximum number of inputs is reached. The single rail gates were then replaced with their dual-rail equivalents and each cluster was then flattened to determine the minterms for the entire function block. The figures show a sample of function blocks from four different clustered implementations of the benchmarks with cluster sizes of 3, 5, 4

Circuit					Non Opt Arc	Opt Arc												Desynchronised	Execution Time (s)
Cluster Size	Name	Cluster Index	Inputs	Outputs	Total	Best Total	Decrease (%)	On-set	Off-set									Total	
									General			Subset			AND				
									Literals	C	AND	Literals	C	AND	Literals	C	AND		
n/a	drfulladder	-	6	4	96	60	37.5	40	20	8	6	20	10	4	24	8	6	-	0.15
	1of4fulladder	-	10	6	384	216	43.8	144	72	32	20	84	44	8	96	32	20	-	0.25
	3of6fulladder	-	14	8	44800	17806	60.3	15175	2631	615	1629	2819	897	1347	5737	870	1374	-	537
3	C6288	131	6	6	144	35	75.7	22	15	7	4	13	8	3	17	8	3	32	0.01
	C6288	871	6	4	144	39	72.9	22	9	4	6	13	8	3	17	8	3	64	0.01
4	C6288	173	8	10	640	79	87.7	62	17	8	16	18	16	8	38	15	9	72	0.23
6	C6288	52	12	14	5376	173	96.8	139	34	16	25	31	23	18	62	20	21	152	2.79
	C6288	43	12	12	3840	162	95.8	118	53	19	16	44	25	10	77	25	10	204	1.85
8	C6288	140	16	18	36864	247	99.3	202	50	22	34	45	31	25	97	28	28	264	663
	C6288	144	16	16	32768	275	99.1	213	62	23	34	68	39	18	132	37	20	272	699

Table 1: Results of Prime Indicant Synthesis

and 8 dual-rail inputs (6,8,12 and 16 function block inputs).

The results of the ISCAS blocks show a dramatic reduction in the literal count of function blocks (up to 99.3%) compared to the canonical architecture, particularly in large function blocks with many outputs where the indication may be distributed between the outputs. The results for the *m*-of-*n* encoded designs are less dramatic showing a reduced literal count of between 37-60%. The results demonstrate the ability of the synthesis procedures to reduce the specification of indicating function blocks. Prior to this work, constructing an efficient indicating implementation for an arbitrary function block was very difficult without resorting to conventional synthesis tools and desynchronisation.

A comparison with the desynchronisation approach of Zhou [19] is also made. Zhou's input relaxation technique was applied to the full benchmark and the resulting netlists were then clustered. The results show very similar literal count figures for both techniques; however, the implementations are quite different. The desynchronised versions are all multi-level implementations, whereas the prime indicant covers are all two-level. It is expected that when combined with the indicating decomposition techniques the size of prime-indicant implementations will reduce still further.

The execution times of the synthesis procedures are given in the final column of table 1. It is clear that the execution time is exponential in the number of inputs. This is due to the implementation of the indicant cover algorithm described in 5.1, where 2^n implicants are generated from n raised literals. A better implementation of this algorithm is the subject of further research.

7. Conclusions

This paper presents algorithms for the optimisation of indicating function blocks, by distributing don't cares amongst the outputs of a function block. An architecture that allows any indicatable function block to be implemented was presented along with the restrictions necessary for it optimised. The algorithms show dramatic improvements (up to 99% in some cases) in the two-level implementation of large function blocks.

Further research to develop more efficient algorithms to create mutually-exclusive indicant covers and off-set optimisation will be undertaken. As well as a study of the effectiveness of using SAT solvers to solve the reduction set selection problems. More complicated costing models, which will target testability and use performance modelling will also be explored.

8. Acknowledgements

The authors would like to thank the reviewers for their helpful comments and Michael Theobald for his insight and discussions that were invaluable in producing the final version of the paper.

9. References

[1] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen , G. D. Hachtel, "Logic Minimization Algorithms for VLSI Synthesis",*Klewer*,1984.

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

[3] T. Chelcea, G. Venkataramani, S. C. Goldstein , "Area Optimizations for Dual-Rail Circuits Using Relative-Timing Analysis", *Proc ASYNC-07*, 2007.

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

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

[6] S. Khatri, S. Sinha, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "SPFD-Based Wire Removal in Standard-Cell and Network-of-PLA Circuits", *IEEE Trans. CAD,* v. 23(7), 2004

[7] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, A. Yakovlev: "Basic Gate Implementation of Speed-Independent Circuits", *Proc. DAC-94*, 1994.

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

[9] S. Malik, L. Lavagno, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Symbolic Minimisation of Multilevel Logic and the Input Encoding Problem", *IEEE Trans CAD*, v 11(7), 1992.

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

[11] 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.

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

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

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

[15] M. Theobald, S. M. Nowick, "Fast Heuristic and Exact Algorithms for Two-Level Hazard-Free Logic Minimisation" , *IEEE Trans CAD*, v.17(11), 1998.

[16] W. B. Toms. "Synthesising Quasi-Delay-Insensitive Datapath Circuits", *PhD Thesis*, University of Manchester, 2006.

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

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

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