M-of-N Code Decomposition for Indicating Combinational Logic

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. In particular, mapping large function blocks into cell-libraries is difficult as decomposing gates introduces new signals which may violate indication. This paper presents a novel method for implementing any m-of-n encoded function block using "bounded gates", where any gate may be decomposed without violating indication. This is achieved by successively decomposing the input encoding into smaller m-of-n codes. The method described in the paper uses algebraic extraction techniques to efficiently determine and quantify potential re-encodings. The results of the synthesis procedure are demonstrated on a range of combinational function blocks.

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 [3], becomes complex. Self-timed circuits [15], 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. Self-timed circuits use a procedure called *indication*, where the outputs of the circuit signal to the environment that the internal signals of the circuit are in a steady state and the circuit is ready to accept more input.

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 using an unordered (or DI) code. Furthermore, implementing large combinational logic functions is complex as decomposing gates introduces new signals which need to be indicated by the outputs. Existing self-timed decomposition methods [1][9], designed for control circuits, struggle with the high concurrency and large amounts of sharing between functions in combinational logic.

This paper presents a novel, automated, method to implement any *m*-of-*n* encoded combinational-logic function block using *bounded-gates*, where gates may be arbitrarily decomposed without violating indication (and so the number of inputs of all gates can be bounded to any value). This is achieved by successively decomposing the input encoding into smaller *m*-of-*n* codes. The paper first defines the requirements for bounded gate implementation of indicating functional blocks. A method of decomposing *m*-of-*n* codes using *unordered divisor-sets* is then presented. The method is shown to allow the implementation of *any* *m*-of-*n* encoded function-block using only bounded gates. The paper then describes a technique to reduce the cost of divisor-sets using algebraic extraction to identify potential encodings from the sharing between code-words. The technique can also be used to select encodings based on the structure of the functions in function-blocks as well as the encoding of the inputs. The results of the technique are presented on a range of *m*-of-*n* encoded function blocks.

1.1 Existing Synthesis Approaches

A popular method of constructing self-timed datapath circuits is an approach called *desynchronisation* [10][3], 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 an equivalent self-timed implementation. This approach allows large self-timed datapaths to be constructed relatively easily, however the overhead of indication is high as each signal needs to be encoded in a dual-rail code. 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* [5][21] and *relative timing* [4].

More recently, a block-level approach to synthesis has been proposed [7], where datapaths are constructed from function-blocks connected by encoded channels. This can reduce the cost of implementing self-timed datapaths by distributing the cost of indication between the outputs of a function block. A synthesis technique to create optimal implementations of arbitrary-encoded function blocks (using unbounded gates) was presented in [17]. However, effective exploitation of the block-level approach is constrained due to limitations on the size of function-blocks as large function blocks cannot be mapped to physical libraries. The techniques presented in this paper are designed to be used within a block-level synthesis framework to deconstruct large function-blocks, before optimisations are applied.

1.2 Decomposition Techniques

Several techniques for technology-mapping and decomposition of Speed-Independent and QDI control circuits have been proposed. Burns [1] analyses the conditions for decomposition of individual sequential elements without sharing. Kondratyev [9] presented a technique based on analysing the effect of signal insertions on state-graphs. In Kondratyev's techniques the indication of signals can be shared between multiple signals, and so shared divisors of functions within a circuit can be used. *Progress conditions* define the additional cost a candidate decomposition will have on other functions in the circuit. A candidate divisor will be rejected if it increases the cost of other function implementations by more than a single literal.

Control circuit decomposition techniques struggle when applied to combinational logic circuits: The large number of concurrent signals, results in an exponential number of states in the state-graph. The expensive checks required to ensure each divisor does not violate speed-independence are largely redundant because of the simple sequential behaviour of the circuits. The large amount of sharing between functions means that decomposing a function will affect any further decompositions of other functions. Progress conditions determine how expensive the current decomposition will be but give no indication of how other decompositions may be affected.

The techniques presented in this paper decompose circuits by re-encoding their inputs. Multiple decompositions are implemented concurrently, removing the need to check each candidate divisor. The techniques generalise existing, manual, techniques for implementing indicating combinational logic. DIMS [16] techniques, where the minterms of a function block are shared by the output functions, effectively re-encodes the inputs of the function-blocks into a one-hot code. Fant [5] suggested decomposing 1-of-*n* function blocks by creating *sum* and *product partitions*. Like DIMS, the product partitions create a one-hot code from the concatenation of several input code groups within a function block. Sum partitions combine code-words that appear together in functions to reduce the number of literals in the function block. The techniques presented in this paper automate the generation of sum and product positions for 1-of-*n* codes and provide a method to determine of the cost of each on the function block. Furthermore, they extend the concept to *m*-of-*n* codes and can actually create partitions *within* code-groups.

1.3 Hazard-Free Combinational Logic Synthesis

A related topic is that of hazard-free logic synthesis [13]. 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* (MIC): where the inputs transition from one function minterm to another. Multi-level synthesis techniques for hazard-free logic have been explored by Nowick [14], who describes the conditions necessary for a multi-level hazard-free implementation of a function and a decomposition method to implement the function in a canonical form. Furthermore, the algebraic techniques used in this paper were identified by Kung [11] as being hazard-non-increasing optimisations. The hazard-freedom of multi-level circuits is preserved providing each sub-circuit preserves the hazard-freedom of the original specification. In multi-level indicating logic, it is not enough to preserve the indication of the original circuit. The decomposition introduces new signals which must be also be indicated along with the input transitions.

2. Indicating Combinational Logic

2.1 Definitions

- A literal is a variable or its negation, v_1 or $\overline{v_1}$.
- A cube is a set of literals C where $v_i \in C \Rightarrow \overline{v_i} \notin C$. A cube represents the function which is a conjunction of its literals.
- The *width* of a cube is the number of literals it contains, denoted $|C|$
- An *expression* is a set of cubes. An *irredundant* expression is an expression where no cube is a proper subset of another.

Cube $\{v_1, v_2, v_3\}$ is written as $v_1v_2v_3$, and expression $\{\{v_1, v_2\}, \{v_3\}\}$ is written as $v_1v_2 + v_3$. However, set operations on cubes and expressions are different to Boolean operations on the functions they represent. For example $\{v_1, v_2\} \subseteq \{v_1\}$, but the logic function v_1 is not contained in v_1v_2. In this paper all operations on cubes refer to set operations rather than Boolean functions.

- The *support* of an expression, *sup*(*f*), is the set of variables where:

$$sup(f) = \bigcup_{p \in f} p$$

  Two expressions, *f* and *g*, have *disjoint support* if $sup(f) \cap sup(g) = \varnothing$.

- The *product* of two expressions *f* and *g* is the irredundant expression:

$$fg = \{c_i \cup d_i | \ c_i \in f, d_i \in g\}$$

  If *f* and *g* have disjoint support then *fg* is called the *algebraic product*.

- Expression *g divides* expression *f* (f/g). If *f* can be rewritten in the form:

$$f = qg + r$$

where $q \neq \emptyset$. If qg is an algebraic product then g is an *algebraic divisor* of *f*. *g* divides *f evenly* if $r = \emptyset$.

2.2 Self-Synchronising Code Systems

In indicating logic circuits, the circuit variables form *code systems* which describe their behaviour during circuit operation. A code is a tuple (*V,Z,A*) where:

- $V = \{v_1, \ldots, v_n\}$ is a set of binary variables.
- $Z = \{z_1, \ldots, z_s\}$ is a set of values of V
- $A = \{(z_1 – z_2), \ldots, (z_{s-1} - z_s)\}$ is a set of *Allowed-Transition-Sets (ATS*) which describe transitions between values.

Each 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. Each ATS, (*a-b*), is defined by two cubes:

- A *transition constant term* – which contains all the variables that do not transition in the ATS (This is equivalent to the *transition cube* of a MIC)*:*

$$\omega(a, b) = \{b_j | \; a_j = b_j\}$$

- A *transition variation term* – which contains the final values of the variables that transition within the ATS:

$$\varepsilon(a, b) = \{b_j | \; a_j \neq b_j\}$$

In *two-phase* code systems, all ATS occur between a data value from the set, *D*, of all valid data values and a spacer from the set of all spacer values, *S*, and so $Z = D \cup S$. In this paper only *return-to-zero* (RTZ) code systems with a single spacer value $s = \{v_1 = 0, \ldots, v_n = 0\}$ are considered. In single-spacer code-systems, the transition variation term of each spacer-to-data transition is unique. Therefore, the data values of a code system can be represented by a set of cubes, called *code-words*, that correspond to the transition variation terms of all spacer-to-data transitions.

A code system is called a *Self-Synchronising Code system* (*SSC*) if the termination of an ATS can be determined by the value of the variables in *V*. In order to be an SSC, the set of code-words must form an *unordered* (or Delay-Insensitive) code [20]:

- A code is *unordered* if for each pair of code-words, d_i and d_j:

$$d_i \subseteq d_j \Rightarrow d_i = d_j$$

- The *weight* of code-word *d, w(d),* is:

$$w(d) = \sum_{j=1}^{n} \alpha_j$$

where:

$$\alpha_j = \begin{cases} 0 \quad if \quad p_j = \overline{v_j} \quad or \quad p_j = * \\ 1 \quad if \quad p_j = v_j \end{cases}$$

In an RTZ SSC, the weight of each code-word is equivalent to the width of the cube: $w(d_j) = |d_j|$

- If two code-words, d_i and d_j, share a common cube, *c*, then the quotients of *c* in each code-word, q_i and q_j, are called the *differentials* of d_i and d_j.Where:

$$q_i \neq \emptyset, \; q_j \neq \emptyset \text{ and } (q_i \not\subset q_j) \wedge (q_j \not\subset q_i)$$

If $q_i \cap q_j = \emptyset$, *c* is the *maximal* common cube of d_i and d_j.

In this paper we consider only one specific class of codes, *m-of-n* codes:

- An *m-of-n* code is an unordered code where each code-word has weight *m*.
- The size of an *m*-of-*n* code is *n* **choose** *m*, denoted by C_m^n :

$$\frac{n!}{m!(n-m)!}$$

2.3 Indicating Combinational Logic

Indicating combinational logic function blocks consist of two self-synchronising code systems, an input code system *X* and an output code system *Y*. An ATS (*a-b*) on code system X causes a subsequent ATS (*k-l*) on code system *Y*. The operation of the block is determined by two multi-valued functions:

- $Y = F(X)$ which maps data values $D^X \in X$ to data values $D^Y \in Y$
- $Y = G(X)$ which 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 $G(s^X) = s^Y$ for all function blocks.

Example 2.1 *A* is a combinational logic block. The input code system *X* consists of four variables, $X = \{x_1, x_2, x_3, x_4\}$, and contains four 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:

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

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

The code words of D^X are:

$$D^X = \left\{ d_1^X = x_1x_3, d_2^X = x_2x_3, d_3^X = x_1x_4, d_4^X = x_2x_4 \right\}$$

The output code system, *Y*, consists of two variables, $Y = \{y_1, y_2\}$, and contains 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:

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

The code words of $D^Y$are:

$$D^Y = \{d_1^Y = y_1, d_2^y = y_2\}$$

$F(X)$ is defined as follows:

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

$$d_2^Y = d_4^X$$

Varshavsky[19] defined the requirements for an indicating implementation to be constructed for a function block:

Definition 2.1 *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:

$$\forall d_i^X \in D^X, \exists d_i^Y \in D^Y \Big| \; d_i^Y = F(d_i^X)$$

$$s^Y = G(s^X)$$

In order for a function block to be indicating, the transitions of individual input variables must be *translated* by the output functions of the function block. A transition is translated by an output in an ATS if the output cannot transition until the transition has occurred. Within each ATS not all input transitions are directly translated to the output of a function, as this would mean some function must change transition 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.

Definition 2.2 *In order for a function-block to be indicating, at least one output must translate each input transition in every ATS of the input code-system.*

2.4 Canonical Architecture

The canonical architecture shown in figure 1 forms the basis of several indicating logic styles such as DIMS [16] and NCL-D [10] and the optimised architecture of [18]. In the canonical architecture, the code-words of the input code system are implemented using a Muller C-element. As the code-words of X are unordered, they are mutually-exclusive within the code-system. Each C-element can only transition once all of the inputs in the ATS have transitioned and all of the input transitions are translated. The sequential behaviour of the C-element translates all of the transitions on both spacer-to-data and data-to-spacer transitions. Therefore, providing C-elements are used, data-to-spacer ATS can effectively be ignored.

The indication of both the canonical and optimised architectures relies on being able to implement each code-word in a single gate. If the width of a code-word is larger than the C-elements in the target library, then the code-words need to be decomposed. The remainder of this paper describes a decomposition method that will allow any m-of-n encoded function block to be implemented using *bounded C-elements* without violating indication. The methods defined in this paper decompose a function-block into several function blocks. Each function block may be implemented using either the canonical architecture or some other function block optimisation technique. As the actual

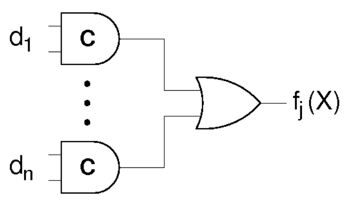


Figure 1: Canonical Architecture

implementation of the individual function blocks is not known at the time of decomposition, no attempt to target real technology libraries was made and physical technology-mapping needs to take place during the implementation phase.

3. Decomposing M-of-N codes

3.1 Bounded C-element Implementation

In order to implement the canonical architecture using bounded C-elements, each C-element may be decomposed in any manner. However, to maintain indication, the transitions of all decomposed C-elements must be translated by the outputs of the function block. As the cubes of the decomposed C-elements are smaller than the original C-element cube, they may be contained within other code-words. Therefore, the C-element will transition during the ATS corresponding to the other code-words and all of the transitions of the C-element are not translated. A solution to this problem is to re-encode the input-code system to ensure that any decomposed C-elements do not contain any other code-words. This removes the need for expensive checking of candidate decompositions as each C-element can be decomposed into smaller C-elements of any number of inputs, without violating indication.

Theorem 3.1 *A code system may be implemented in the canonical architecture using bounded gates if and only if for each pair of code-words, d_i^X and d_j^X, in the input code system:*

$$\left| d_i^X \cap d_j^X \right| \le 1$$

Proof If each decomposed C-element implements a divisor of d_i^X then the C-element tree will translate all input transitions in the ATS $(s^X - d_i^X)$ and $(d_i^X - s^X)$.

If $\left| d_i^X \cap d_j^X \right| \le 1$, then no sub-cube of d_i^X with a width of 2 or more can be contained in d_j^X. Therefore, provided each C-element contains two or more distinct inputs, each C-element will transition only during ATS $(s^X - d_i^X)$ and $(d_i^X - s^X)$. □

3.2 Divisor Sets

In order to implement technology-independent function blocks, the input code system must be re-encoded to fulfil the properties of theorem 3.1. A new code system, W, is created and the function block is re-mapped to operate on code system W rather than X. To create an indicating implementation for the function block, definition 2.1 must be upheld: code system W must be an SSC and each ATS of code system X must map directly to an ATS of W.

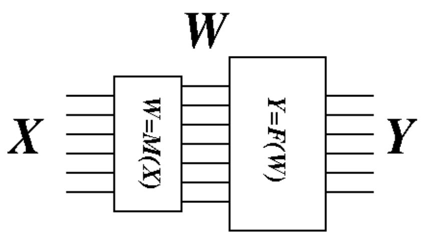


Figure 2: Function block Decomposition

To implement code system W a new function block is created defined by the multi-valued function $W = M(X)$, as shown in figure 2. To reduce the number of shared literals in X we must construct the variables of code system W from common cubes of code-words of X. The variables of W form a set of divisors, U, of the code-words of code system X. In order to form an SSC we must select the divisors in such a way that U forms an "*unordered cover*" of the code-words of X. It may not be possible to construct a code system that fulfils the properties of theorem 3.1 from sub-cubes that may be implemented using bounded gates themselves and therefore W may need to be further decomposed.A set of divisors fulfilling these properties is called an *unordered divisor set*.

Definition 3.1 *An unordered divisor set of code system X is a set of divisors, U, of the code-words of X such that:*

i Each pair of divisors is unordered:

$$\forall c_i \in U, c_j \in U| \quad (c_i \subseteq c_j) \Rightarrow c_i = c_j$$

ii For each $d_j^X \in X$, the set $U_j \subseteq U$ contains the divisors of d_j^X:

$$U_j = \{c| \quad d_j^X / c\}$$

For each pair of ATS in code system X the associated divisor sub-sets must be unordered:

$$\forall d_i^X \in X, d_j^X \in X| \quad U_i \subseteq U_j \Rightarrow U_i = U_j$$

The following two theorems show how divisors sets can be constructed and used to implement any m-of-n code using bounded gates:

Theorem 3.2 *An unordered divisor set may be constructed from the set of k-width sub-cubes of an m-of-n code.*

Proof In an m-of-n code if two code-words, d_i and d_j, share a maximal common cube, c, then each code-word contains a differential of width $m - |c|$. There are C_k^m sub-cubes of each code-word with width k and so for each code-word there will be $C_k^m - C_k^{|c|}$ sub-cubes of each word that intersect with the differential and so cannot be contained in the other code-word. Therefore the set of divisors U_i and U_j are unordered. As each divisor is of width k, the divisors form a k-of-n code and the divisor-set is unordered. □

Theorem 3.3 *A technology independent implementation for any m-of-n code can be created by recursively applying a m-*1 *width divisor set.*

Proof As each code-word in an m-of-n code is unordered, the width of largest common cube that can be shared between two code-words is m-1. For each code-word in an m-of-n code there are:

$$C_{m-1}^m = m$$

sub-cubes of width m-*1*. If c_i and c_j share a m-1 width common cube, then there will be

$$C_{m-1}^m - C_{m-1}^{m-1} = m-1$$

sub-cubes of each cube that intersect with the differential of each cube. Therefore, the two code-words share only one m-1 width sub-cube and the technology-independent properties described in theorem 3 are upheld.

As all the sub-cubes are m-1 width, they form an m-1-of-n code which may be decomposed using m-2 divisors in the same way. Therefore it is possible to construct a technology-independent implementation for any m-of-n encoded function block. □

Once the divisor set is determined the original functions may be "re-mapped" by substituting the literals in the function cubes corresponding to each divisor with a literal from the new code system. The substitution needs to be done concurrently, as the divisors may share literals that will be removed in substitution.

Example 3.1 A 3-of-5 code system contains the variables $X = \{x_1, x_2, x_3, x_4, x_5\}$ and the code-words:

$$d_1^X = x_1x_2x_3, d_2^X = x_1x_2x_4, d_3^X = x_1x_2x_5, d_4^X = x_1x_3x_4$$

$$d_5^X = x_1x_3x_5, d_6^X = x_1x_4x_5, d_7^X = x_2x_3x_4, d_8^X = x_2x_3x_5$$

$$d_9^X = x_2x_4x_5, d_{10}^X = x_3x_4x_5$$

The set of 2-of-5 sub-cubes of the code-words are:

$$w_1 = x_1x_2, w_2 = x_1x_3, w_3 = x_1x_4, w_4 = x_1x_5, w_5 = x_2x_3$$

$$w_6 = x_2x_4, w_7 = x_2x_5, w_8 = x_3x_4, w_9 = x_3x_5, w_{10} = x_4x_5$$

The sub-cubes form an unordered divisor set, and hence an SSC. The code-words of code system W are:

$$d_1^W = w_1w_2w_5, d_2^W = w_1w_3w_6, d_3^W = w_1w_4w_7, d_4^W = w_2w_3w_8$$
$$d_5^W = w_2w_4w_9, d_6^W = w_3w_4w_{10}, d_7^W = w_5w_6w_8, d_8^W = w_5w_7w_9$$
$$d_9^W = w_6w_7w_{10}, d_{10}^W = w_8w_9w_{10}$$

4. Reduced Divisor Sets

Theorems 3.2 and 3.3 demonstrate that technology independent implementations of any m-of-n code can be created using unordered divisor sets. However the cost of implementing a code in this way can be expensive. If the m-of-n code is reduced or consists of smaller m-of-n codes concatenated together it may not be necessary to use all of the k-width sub-cubes of code-words and unordered divisor sets may be constructed from a subset of possible divisors. In the remainder of this paper we present efficient methods to find reduced divisor sets that may be used to decompose m-of-n codes.

4.1 Differentials

In order to reduce the number of shared literals between the code-words of the code system, the variables of code system W

are formed from common cubes of code system X. A divisor set, U, is unordered if each pair of subsets, U_i and U_j (containing the divisors of code-words d_i and d_j respectively), are not properly contained in each other. If d_i and d_j share a common divisor, c, then U must contain *two further divisors*, that intersect with the differential of each code-word to ensure that the sets U_i and U_j are not subsets of each other.

The divisors in an unordered divisor set have two purposes: to reduce sharing between code-words by extracting common cubes and to *differentiate* code-words that share divisors. In order to make a divisor set unordered, it may be necessary to add non-shared divisors to the set to differentiate code-words. However, if the quotient, q, of divisor c, in a code-word is also a common cube, q may be used to differentiate other divisors as well. Therefore, the size of divisor sets can be reduced by selecting divisors that share quotients with other divisors in the set.

Example 4.1 A reduced 4-of-7 code system contains the variables $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ and the code-words:

$$d_1^X = x_1x_2x_5x_6, d_2^X = x_1x_2x_5x_7, d_3^X = x_3x_4x_5x_6, d_4^X = x_3x_4x_5x_7$$

If a divisor set is created using the common divisors: $w_1 = x_1x_5$ and $w_2 = x_3x_5$. Then 4 additional divisors are required to differentiate all the code-words:

$$w_3 = x_2x_6, w_4 = x_2x_7, w_5 = x_4x_6, w_6 = x_4x_7$$

and the code-words of code system W are:

$$d_1^W = w_1w_3, d_2^W = w_1w_4, d_3^W = w_2w_5, d_4^W = w_2w_6$$

If however, a divisor set is created from common divisors: $w_1 = x_1x_2$ and $w_2 = x_3x_4$. The quotients of these cubes are also shared cubes and only two additional divisors are required to differentiate all code-words:

$$w_3 = x_5x_6, w_4 = x_5x_7$$

and the code-words of code system W are:

$$d_1^W = w_1w_3, d_2^W = w_1w_4, d_3^W = w_2w_3, d_4^W = w_2w_4$$

4.2 Kernel Extraction

The algorithms to find the reduced unordered divisors sets of a code system aim to determine sets of divisors that share quotients. In order to achieve this kernel-extraction techniques are employed. Kernel extraction was developed by Brayton [2] as an efficient way to determine the multiple-cube divisors of a set of expressions:

- The primary divisors of an expression *f, PD(f)* is the set:

  $$PD = \{f/c | \ |c| = 1\}$$

- An expression is *cube-free* if no single cube divides *f* evenly.
- The kernels of expression *f*, *K*(*f*), is the set of *cube-free primary divisors* of *f*. The expression *c* where $k = f/c$ is called the *co-kernel.*
- Two expressions, *f* and *g*, share a common *multi-cube* divisor if and only if there exists a $k_f \in K(f)$ and $k_g \in K(g)$ such that $|k_f \cap k_g| \geq 2$.

Kernel intersections exploit the fact that in irredundant expressions each multiple-cube divisor must contain a cube-free core. Therefore only the cube-free divisors of expressions, the kernels, need to be explored and the search space is reduced.

$$F = af_1 + bf_2 + ag_3 + cg_4 + ade_5 + bde_6 + cde_7$$
$$G = af_8 + bf_9 + ace_{10} + bce_{11}$$

	Co-kernel	Kernel
F	a	de + f + g
F	b	de + f
F	de	a + b + c
F	f	a + b
F	c	de + g
F	g	a + c
G	a	ce + f
G	b	ce + f
G	f	a + b
G	ce	a + b

		a	b	c	ce	de	f	g
F	a	•	•	•	•	5	1	3
F	b	•	•	•	•	6	2	•
F	de	5	6	7	•	•	•	•
F	f	1	2	•	•	•	•	•
F	c	•	•	•	•	7	•	4
F	g	3	•	4	•	•	•	•
G	a	•	•	•	10	•	8	•
G	b	•	•	•	11	•	9	•
G	f	8	9	•	•	•	•	•
G	ce	10	11	•	•	•	•	•

Figure 3: Co-kernel Cube Matrix for an example network

Rudell [12] presented an efficient approach to determining the kernel-intersections of a network, based on finding *rectangles* within *matrices*:

- A *matrix* is a two-dimensional grid, B, where $B_{ij} \in \{\emptyset,1\}$.
- A *rectangle* is a pair of sets of (not necessarily adjacent) rows and columns (R,C) of matrix B, such that $B_{ij} \neq \emptyset$ for all $r_i \in R$ and $c_j \in C$.

Kernel intersections can be identified by constructing a *co-kernel cube matrix* from the kernels of a network. In a co-kernel cube matrix (figure 3); there is a row for each co-kernel, ck_i, of each expression and its corresponding kernel, k_i. The columns of the matrix represent the unique cubes, kt_j, of all the kernels. The cubes of each function, c_l, are numbered and $B_{ij} = l$ (instead of 1) if $ck_i \cdot kt_j = c_l$. Rectangles in this matrix correspond to kernel intersections within the network.

Rudell's algorithm's assign *values* to rectangles in the matrix and the highest value rectangle selected for extraction. The value of each rectangle is the reduction in literals that substituting the cube into the network will bring and is defined by the number of literals covered by the rectangle minus its *weight*:

$$v(R, C) = \sum_{i \in R, i \in C} V_{ij} - w(R, C)$$

where V_{ij} is the number of literals in the cube covered by (i,j). The weight of a rectangle is the cost of substituting into the network and is given by:

$$w(R, C) = \sum_{i \in R} wr_i + \sum_{i \in C} wc_j$$

where wr_i represents the number of literals + 1 in the co-kernel associated with row r_i and wc_j the number of literals in the cube associated with column j.

4.3 Intersecting Rectangles

Kernel-intersections are used within conventional synthesis routines as a method of reducing the search space when determining multiple-cube divisors. In the techniques described in this paper kernel-intersections are used in a different way: *to determine sets of divisors that share quotients*.

The co-kernels of a code system represent the largest common cubes shared between the code-words of the code-system. The cubes of each kernel represent the quotients of the co-kernel within the code system. Therefore if two co-kernels share a kernel-intersection, the co-kernel forms a common-cube whose quotient is shared. Reduced divisor sets can be constructed by finding sets of cubes whose quotients are shared. This results in two divisor sets each of whose members will differentiate the divisors in the other. These sets form *Intersecting Rectangles* in a co-kernel matrix constructed from the code-system

Definition 4.1 An *intersecting rectangle, IR, of a matrix is a pair* (R^n,C^n) *of sets of rows and columns where for each row* r_i (*column* c_i) *of* R^n (C^n), the set C^n (R^n) *contains all intersecting columns* c_j *(rows* r_j*) where* $B_{ij} \neq 0$.

As they all share quotients, all of the cubes in each set have the same width. If the width of the cubes in R^n is *k*, the width of cubes in C^n will be *m-k*.

Figure 4 shows the co-kernel cube matrix of an example code-system. The code system is constructed from a 2-of-4 code-group (variables *a*0-*a*3) and a 1-of-2 code group (variables *b*0-*b*1). The intersecting rectangles are all shaded on the matrix. There are 4 IRs, which form two sets of transposed rectangles. As both co-kernels and kernel-cubes are shared the *transpose* of rectangle (R^n,C^n), (C^n,R^n), is also an IR in the matrix. The rectangles *A* and *B* correspond to a partition of the two code groups. In rectangle A:

$$R^n = \{a01a1, a0a2, a0a3, a1a3, a2a3\}$$

$$C^n = \{b0, b1\}$$

In rectangles *C* and *D*, the IRs actually decompose the 2-of-4 code groups and create sub-cubes of the code-words. In rectangle *C*:

$$R^n = \{a0, a1, a2, a3\}$$

$$C^n = \{a0b0, a1b0, a2b0, a3b0, a0b1, a1b1, a2b1, a3b1\}$$

In this example all four IRs are unordered divisor sets and maybe substituted into the network. However, not all IRs form unordered divisor sets and there are some checks that need to be performed on each IR.

4.3.1 Complete Rectangles

In order to reduce the search space of the algorithm, co-kernels and kernels are used instead of primary divisors. In an unordered code system the kernel-cubes of a co-kernel correspond to all of the instances of the co-kernel within the code system. However, the intersections of a kernel-cube may not correspond to all of the instances of the kernel-cube within the code and the kernel-cube is not a co-kernel itself. There are two reasons the

Co-Kernels	a0	a1	a2	a3	b0	b1	a0 b0	a1 b0	a2 b0	a3 b0	a0 b1	a1 b1	a2 b1	a3 b1	a0 a1	a0 a2	a1 a2	a0 a3	a1 a3	a2 a3
a0								0	1	3		6	7	9						
a1							0		2	4	6		8	10						
a2							1	2		5	7	8		11						
a3							3	4	5		9	10	11							
b0															0	1	2	3	4	5
b1										C					6	7	8	9	10	11
a0b0		0	1	3																
a1b0	0		2	4														B		
a2b0	1	2		5																
a3b0	3	4	5			D														
a0b1		6	7	9																
a1b1	6		8	10																
a2b1	7	8		11																
a3b1	9	10	11																	
a0a1					0	6														
a0a2					1	7														
a1a2					2	8		A												
a0a3					3	9														
a1a3					4	10														
a2a3					5	11														

Figure 4: Co-kernel Cube Matrix of a 2of4/1of2 Code System

quotient, ck_i, of a kernel-cube, kt_j, in code-word d_l may not be a co-kernel and hence not in the cube-literal matrix:

i ck_i is not contained in any code-word other than d_l.

ii The quotients of ck_i within the code system (including kt_j) are not cube-free.

In order to ensure that an intersecting rectangle is *complete*, the quotients of all of the kernel-cubes in C^n need to be checked and any missing divisors added to R^n. Once the IR is complete, R^n and C^n form two sets of cubes where all the quotients of the cubes in one set are contained within the other set. If a complete IR does not cover a code-word, c_i, of the code system, then the IR cannot be an unordered divisor set as $U_i = \varnothing$ and hence is contained in every other subset of *U*.

4.3.2 Types of Intersecting Rectangle:

As both the rows and columns of the intersecting rectangle are shared, each IR has a transposed rectangle corresponding to the divisor set generated when the shared cubes of the columns form co-kernels themselves. There are two types of intersecting rectangle:

- *Distributed*: The cubes of R^n and C^n do not differentiate every term independently and must be combined in order to construct an full unordered divisor set. An IR is distributed if:

  $$\forall ck_i, ck_j \in R^n, d_l^X \in X| \quad ck_i \cup ck_j \neq d_l^X$$

  where $ck_i \cup ck_j$ is the set union of the literals in cubes ck_i and ck_j.
- *Combined*: The cubes of both R^n and C^n differentiate all of the code-words and each set forms an unordered divisor set. An IR is combined if:

  $$\exists ck_i, ck_j \in R^n, d_l^X \in X| \quad ck_i \cup ck_j = d_l^X \quad \text{if } k \geq m/2$$

  $$\exists kt_i, kt_j \in C^n, d_l^X \in X| \quad kt_i \cup kt_j = d_l^X \quad \text{otherwise}$$

Theorem 4.1 *If an IR is a complete combined intersecting rec-*

tangle then the co-kernels of the IR form an unordered divisor set.

Proof If (R^n,C^n) is a complete rectangle, then there exists an equivalent complete rectangle (C^n,R^n), therefore without loss of generality we assume $k \geq m/2$. The IR is combined if there exists two co-kernels, ck_i and ck_j, whose union is equal to code-word d_l^X. As the IR is complete, there exists two kernel-cubes in the IR, kt_p and kt_q, where

$$ck_i \cdot kt_p = ck_j \cdot kt_q = d_l^X$$

as $ck_i \cup ck_j = d_l^X$:

$$ck_i \subseteq kt_q \text{ and } ck_j \subseteq kt_p$$

The set R^n contains all the quotients of kernel-cubes, kt_p and kt_q. As kernel-cube kt_q is contained in ck_i there must be quotients of kt_q in R^n that are contained in each cube of the kernel k_i associated with ck_i. The cubes of each kernel are the differentials of the code-words containing the co-kernel, therefore, for every code-word that contains ck_i there is some ck_l contained within R^n that differentiates it from every other code-word containing ck_i.

In order for a row or column to be added to the IR it must intersect with some co-kernel or kernel-cube in the IR. For each ck_r that intersects with ck_i over kernel-cube kt_s, there is a cube in R^n that contains kt_s (whose quotient is kt_q) and hence a further kernel-cube that is contained in ck_r. Therefore, as in the previous paragraph, all code-words that contain ck_r are differentiated. The process continues with all intersecting co-kernels and hence the divisor set is unordered. □

A combined rectangle can be viewed as a merge of two transposed intersecting rectangles. As a code-word can be constructed from the union of two co-kernels (or kernel-cubes) the two rectangles intersect each other, and the rectangle contains all of the divisors necessary to ensure it is unordered. Figure 5 shows such a rectangle, the two sub-rectangles are shaded in different shades. The rectangles intersect in code-word d_6.

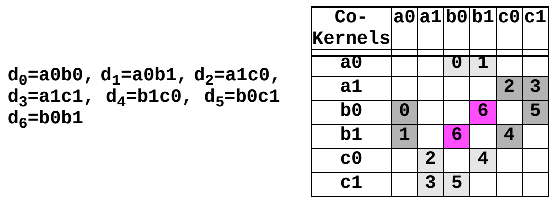
If an IR is not a combined rectangle it is still possible to create an unordered divisor set out of it. However, in order to differentiate all code-words a further divisor set will be needed.

Theorem 4.2 *If an IR is a complete distributed intersecting rectangle then the co-kernels of the IR form an unordered divisor set provided $R^n$and C^n have disjoint support.*

Proof If an IR is distributed then the union of any two co-kernels does not equal any code-word. There are two types of distributed IRs, depending on the relationship between pairs of co-kernels:

Type 1: $\forall ck_i, ck_j \in R^n, d_l^X \in X| \quad ck_i \cup ck_j \not\subset d_l^X$

If the union of any pair of co-kernels is not contained within any code-word, then each code-word in the code system can have at most one divisor in the divisor set. As all code-words are covered by a complete IR, no subset of *U* can be a proper subset of another and so the divisor set is unordered. However, subsets U_i and U_j can be equivalent and in order to differentiate equivalent code-words a further divisor set is necessary.



d_0=a0b0, d_1=a0b1, d_2=a1c0,
d_3=a1c1, d_4=b1c0, d_5=b0c1
d_6=b0b1

Co-Kernels	a0	a1	b0	b1	c0	c1
a0			0	1		
a1					2	3
b0	0			6		5
b1	1		6		4	
c0		2		4		
c1		3	5			

Figure 5: Example Combined Intersecting Rectangle

Type 2 $\exists ck_i, ck_j \in R^n, d_l^X \in X| \quad ck_i \cup ck_j \subset d_l^X$

The union of some pair of co-kernels is contained within some code-word of the code system d_l^X. As $ck_i \cup ck_j$ is a proper subset of d_l^X, there exists a sub-cube of d_l^X that is not in ck_i or ck_j. Therefore, the kernel-cubes kt_p and kt_q, where

$$ck_i \cdot kt_p = ck_j \cdot kt_q = d_l^X$$

are not contained in ck_i and ck_j:

$$ck_i \not\subset kt_q \text{ and } ck_j \not\subset kt_p$$

and, unlike combined IRs, cubes which intersect with the differential of each co-kernel may not be in R^n. As d_l^X has two divisors in the divisor set and the set cannot be guaranteed to be unordered.

Therefore, without explicitly checking the rectangle forms an unordered divisor set only distributed rectangles of type 1 may be employed. If we select a distributed rectangle of type 1, in order to differentiate all code-words, both R^n and C^n will need to be applied to the code system. If R^n and C^n have disjoint support then the divisors of each set form independent code systems that may be decomposed independently. If the support of R^n and C^n is not disjoint then the divisors form a single code system. If $k \neq m/2$, the widths of the divisors in the two set will not be equal and the divisors will not form an *m*-of-*n* code.

If the IR is a type 2 distributed rectangle, then two co-kernels must be contained within a code-word. In order for two co-kernels to be contained within a code-word, the quotient of each co-kernel must contain the non-shared literals of the other co-kernel. Therefore, distributed IRs where R^n and C^n have disjoint support are all type-1 distributed IRs and form independent unordered divisor sets. □

The intersecting rectangles *A* and *B* of figure 4 are distributed, where as IRs *C* and *D* are combined. In general, distributed rectangles with disjoint support will partition variables between code-groups and combined rectangles will decompose the cubes within groups.

Theorems 4.1 and 4.2 determine which IR can be employed to construct unordered divisor sets. However, in order to implement divisor sets correctly, distributed divisor sets must be distinguished from combined sets. It is possible to get a distributed divisor set where R^n and C^n have the same support and so the individual cubes of R^n and C^n must be checked for containment. If a cube of one set contains a cube of another, the IR is combined, if the support of the two sets is not identical or no cube is contained within a cube from the other set the IR is distributed.

As distributed IRs form independent code systems it is not always necessary to use C^n to differentiate code-words if other

IRs with disjoint support are available. Therefore, any IRs with disjoint support may be applied together.

To select between rectangles with overlapping support, a cost function is used which determines the cost, in literals, of adding combined rectangles to the network:

$$\frac{\frac{\sum_{i \in R, j \in C} V_{ij}}{|(R^n, C^n)|}}{n(R^n, C^n)} - w(R^n, C^n)$$

where $n(R^n,C^n)$ represents the number of code-words actually covered, and $|(R^n,C^n)|$ is the number of matrix elements in the IR.

The weight of the rectangle is reflects the fact that, if the IR is combined, several divisors will be substituted in to each original network cube:

$$\sum_{i \in R} wr_i + \sum_{i \in C} (|ec_j| \cdot (|C_l - R_l| + 1))$$

where wr_i is the number of literals in the co-kernel associated with row i, $|ec_j|$ is the number of matrix entries in column j and $|C_l - R_l|$ is the number of literals in the support of column set that are not in the support of the row set.

4.4 Multi-cube Divisor Sets

In combinational logic function blocks, the functions of the output code system are dependent on different code-words of the input code system. In order to reduce the cost of the function-block, divisor sets may be selected in such a way that the code system divisors form multiple-cube divisors of the output functions in the function-block. As kernel-extraction techniques were developed to find multiple-cube divisors, the techniques described in the previous section can easily be modified for this purpose.

A multi-cube divisor introduces an OR-gate into the network. In order to maintain indication within the network, the OR gate must translate the transitions on its inputs (the code system divisors) *and* all of the transitions on the output must be translated. Therefore, we select only divisors which have the same quotients in each function within the function block. Such a divisor is called a *maximal kernel-intersection*.

Theorem 4.3 *If mki is a set of kernel-cubes whose quotient is the same in all dependent functions of the function-block then mki is an indicating multiple-cube divisor.*

Proof In order to be indicating, a multiple-cube divisor must translate all of the transitions on the inputs of the divisor and all of the outputs of the divisor must be translated by output functions:

Output Translation If all the cubes of *mki* share the same quotient in all the functions that are dependent on any cube, then *mki* is a divisor of every dependent function of it cubes and can be substituted in to all of them. The properties of indication mean that for each ATS in an input code system, at least one output will transition and translate the transitions in the input ATS. As *mki* is substituted into every dependent function, for each ATS in which *mki* transitions, some output it is substituted into also transitions and so all transitions on the output of *mki* are translated.

Input Translation: An OR-gate will translate all of its inputs if and only if they are mutually-exclusive [18]. If two cubes, ck_i and ck_j have the same quotients in all dependent functions of the logic block then they must be mutually exclusive. In order for ck_i and ck_j to be non-mutually exclusive, there must be some code-word, d_l^X, of which both cubes are divisors. The quotients of ck_i and ck_j in d_l^X are q_i and q_j respectively. The literals of d_l^X can be partitioned in to four sets:

$$s0 = ck_i \cap ck_j, \; s1 = ck_i - ck_j,$$

$$s2 = ck_j - ck_i, \; s3 = d_l^X - (s0 \cup s1 \cup s2)$$

Therefore:

$$ck_i = s0 \cup s1, \; ck_j = s0 \cup s2$$

$$q_i = s3 \cup s2, \; q_j = s3 \cup s1$$

If ck_i and ck_j have the same quotient in all functions, then there exists a cube of f, d_m^X, where:

$$d_m^X = ck_i \cdot q_j = s0 \cup s1 \cup s3$$

and

$$d_l^X \subset d_m^X$$

This violates the SSC code system and so ck_i and ck_j are mutually-exclusive within code system X. □

As in conventional multi-cube divisor synthesis, the co-kernel cube matrix is constructed from the kernels of the individual functions in the function-block. The unordered divisor sets are constructed from intersecting rectangles within these matrices. A divisor set has a multi-cube divisor if it has a maximal kernel intersection. As the cube-literal matrix is constructed from the kernels of the individual functions of the function-block rather than the whole code system, there may be even more missing kernel-cubes than in single-cube divisor set extraction. When searching for any missing quotients of the kernel-cubes care must be taken that no missing function-cube contain any cubes of the maximal intersections within the rectangle as this will violate the properties of theorem 4.3 and mean that the cubes of the intersection may not be mutually-exclusive within code system X. Once the IR is complete, the checks described in the previous section must be used to determine whether it is an unordered divisor set.

5. M-of-N Code Decomposition Algorithm

An algorithm for *m*-of-*n* code decomposition is shown in figure 6. The algorithm enumerates all possible intersecting rectangles and substitutes as many disjoint rectangles as possible with the highest value. The algorithm only considers the value when differentiating between rectangles with overlapping support rather than the amount of shared literals between the cubes in the divisor set. If the new code system *W* forms an *m*-of-*n* code (i.e. the size of each subset U_j is the same), then code-system *W* can be decomposed to produce a technology-independent implementation. When selecting an intersecting rectangle, the rectangle must first be made complete by adding any missing quotients

```
decompose_function_block(FB) {
   unique_exp = unique(FB);
   kernels = extract_kernels(unique_exp);
   if kernels = Ø then return FB
   construct-cokernel-cube-matrix(kernels);
   intersecting_rects = {};
   while not empty CK-matrix do {
    gen_intersecting_rectangles(CK-Matrix,0,rect)
      rectangle-cost(rect);
      intersecting_rects ∪ rect;}
   if intersecting rects = Ø {
      DS = unique(kernels);
      substitute(FB,DS); return;}
   while (valid_rect = FALSE) {
   max-rectangles= max_value_IR_with_
      disjoint_support(intersecting-rectangles);
   foreach IR in max_rectangles {
      R = rows(IR), C = columns(IR)
      foreach cube in unique_exp {
         Ci = divisors of cube in C;
         for_each c in Ci {
            q = cube/c;
            if q ∉ R then C = C ∪ q }
         Ri = divisors of cube in R;
         if |Ri| ≠ m then discard = true; break;}
      if sup(R) ∪ sup(C) ≠ Ø {
         if sup(R) ≠ sup(C) discard = true;
         foreach r in R {
            foreach c in C {
             if((c ⊆ r) or (r ⊆ c)) contain = true}}
         if contain = false then discard = true;}
      if discard then delete(max_rectangles,IR);
      else {
         substitute(FB,C);
         valid_rect = true;}}}
```

Figure 6: Intersecting Rectangle Selection Algorithm

to the rectangle. This is combined with a check to ensure that the new code-system W is an m-of-n code. If the support of R^n and C^n are disjoint then the rectangle is distributed and no further checks are necessary. However if $sup(R^n) = sup(C^n)$ then the IR must be checked to ensure it is combined. The algorithms to determine the IRs within an extract matrix are very simple. The main overheads of the selection procedure are involved in checking all of the code-words of the code system to ensure the IR is complete and performing a containment check on the cubes of R^n and C^n to determine if the IR is combined. The impact of these checks can be reduced by performing them after the selection process, if a selected IR fails the checks then the selection process must be repeated. After a set of IRs are substituted into the function-block the process is repeated. As all code systems are independent, the entire set of input code systems can be treated as a single expression, when calculating the kernels. The algorithm terminates when there are no further common cubes in the network. If there exists common cubes, but no further intersecting rectangle, then a reduced divisor set is not possible and an unordered divisor set can be constructed from the set of differentials of all common cubes (the kernel-cubes).

The algorithm presented in figure 6, enumerates all of the possible IRs of all widths to minimise the total cost of the decomposed function-block. There are many possible optimisations to this algorithm to reduce the efficiency:

- The kernel extraction process could be limited to k width co-kernels, starting initially at m-1.
- Heuristic rectangle selection algorithms such as PING_PONG [12] could easily be adapted for IRs eliminating the need to enumerate all rectangles.
- Many of the IR checks can be eliminated if rectangle (R^n, C^n) is only selected if the transposed rectangle (C^n,R^n) exists.

The trade-offs between algorithmic complexity and the quality of implementation have yet to be evaluated.

Cluster	Input Encoding	Output Encoding	Total Literal Count	DIMS Literal Count	Decom-posed Literal Count	Multi-cube IRs	Single-Cube IRs	No IR
261 6inputs 5outputs	1of2	1of2	2560	1014	216	3	4	YES
	1of4	1of4	960	566	147	3	1	NO
	2of7	2of7	960	567	489	1	1	NO
	3of6	3of6	1536	760	657	1	2	NO
	4of8	3of7	1536	760	867	1	2	NO
156 8 inputs 5 outputs	1of2	1of2	12800	4598	294	6	6	YES
	1of4	1of4	4608	2550	206	6	2	YES
	2of7	2of7	4608	2551	266	3	2	YES
	3of6	3of6	8192	3576	584	2	4	NO
	4of11	3of7	6144	3061	2603	1	2	NO
174 9 inputs 5 outputs	1of2	1of2	28160	9718	340	6	7	YES
	1of4	1of4	10752	5622	227	7	1	YES
	2of7	2of7	10752	5623	436	4	6	YES
	3of6	3of6	18432	7672	1430	3	2	YES
	5of12	3of7	18432	7672	1430	3	2	YES
106 10 inputs 6 outputs	1of2	1of2	73728	22516	1584	4	8	YES
	1of4	1of4	21504	11252	666	3	2	NO
	2of7	2of7	21504	11253	6921	1	3	NO
	3of6	3of6	36864	15350	1625	3	4	NO
	6of13	4of8	61440	20468	7670	2	3	NO

Figure 7: Decomposition Results

6. Results

Table 7 shows the results of the decomposition algorithm on function blocks generated from the ISCAS benchmark c6288. It is impractical to implement full datapaths using indicating logic synthesis methods, because all of the ATS within the system must be enumerated to ensure the signal is indicating. The function blocks in table 7 were created using a clustering algorithm that traverses a single-rail netlist adding gates to a cluster until a fixed number of inputs is reached. The gates of each cluster are then flattened to create a single truth-table. Each cluster can then be encoded into any m-of-n encoding. The results show the initial literal counts of each cluster in the canonical architecture (assuming unbounded C-elements and 2-input OR gates), the savings due to DIMS minterm sharing and the decomposed literal counts. Also shown are the number of multi-cube intersecting rectangles used, the number of single-cube intersecting rectangles. The final column displays whether or not a non-reduced divisor set had to be used. The results show a large improvement in literal count over both implementations in the canonical architecture and DIMS implementations. However, large function-block implementations are impractical unless optimisations such as those presented in [7] and [18] are employed. The techniques presented in this paper are to be used in conjunction with such block-level techniques in order to allow a much wider range of blocks to be implemented. The incorporation of the decomposition techniques in to a block-level optimisation framework is the subject of future work.

7. Conclusions and Future Work

This paper presents an efficient method to implement any *m*-of-*n* encoded function block using bounded gates. The method decomposes the *m*-of-*n* code system of the inputs, to reduce the sharing between code-words and allow each gate to be decomposed without violating indication. The method decomposes *m*-of-*n* codes by determining *reduced divisor sets*, that are used to construct further *m-of-n* with reduced sharing. An efficient algorithm was presented to determine reduced-divisor sets efficiently. Further research to optimise the algorithm in this paper will be undertaken. In particular heuristic methods may be employed to avoid enumerating all intersecting-rectangles. The decomposition will also be incorporated in to a function-block based self-timed synthesis tool allowing much larger function blocks to be implemented than currently possible.

8. References

[1] S. M. Burns, "General Conditions for the Decomposition of State-Holding Elements", *Proc Async-96*, 1996

[2] R. K. Brayton, C. T. McMullen, "The decomposition and factorization of boolean expressions", *Proc. International Symposium on Circuits and Systems*, 1982.

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

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

[5] K. Fant, "Logically Determined Design", *Wiley*, 2005.

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

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

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

[9] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev,"Logic Decomposition of Speed-Independent Circuits", *Proc of the IEEE*, 87(2), 1999.

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

[11] D. S. Kung. "Hazard-Non-Increasing Gate-Level Optimization Algorithms", *Proc. International Conference on Computer Aided Design*, pp 631-634, 1992.

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

[13] 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.

[14] S. M. Nowick, C. W. O'Donnell, "On the Existence of Hazard-Free Multi-Level Logic", *Proc ASYNC-03*, 2003.

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

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

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

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

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

[20] T. Verhoeff, "Delay-insensitive codes – an overview", *Distributed Computing*, v3(1),1988

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