Self-Timed Realization of Combinational Logic

P. Balasubramanian and D.A. Edwards
School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK
{padmanab, doug}@cs.man.ac.uk

Abstract

A novel synthesis method for self-timed realization of arbitrary combinational logic functions is presented in this paper. The cost of self-timed implementation of a large number of conventional combinatorial benchmarks is provided. A new self-timed system configuration is also proposed in support of the synthesis heuristic that generally favors weakly indicating realizations of combinational logic. The proposed two-level synthesis technique forms a good starting point for the multi-level synthesis of weak-indication circuits and certain preliminary insights in this regard are highlighted.

1. Introduction

'Reliability' has been labeled as one of the five cross-cutting design challenges in the International Technology Roadmap for Semiconductors 2008 update on design [1]. This drives home the point that 'robustness' is becoming an increasing priority for digital logic design in deep submicron technologies. In this scenario, self-timed design attracts increasing interest, as it can inherently tolerate fluctuations in process parameters, temperature and noise [2], whilst guaranteeing correct operation regardless of variations in design components or signal wires. In addition, it features greater modularity. In this perspective, this paper deals with research undertaken in the domain of self-timed combinational logic and presents results corresponding to a new two-level synthesis strategy. The ultimate objective is to arrive at a novel, practically feasible, technology-independent multi-level synthesis strategy for compact weak-indication realization of any combinational logic specification.

The remaining portion of this paper is organized as follows. Section 2 describes a self-timed logic block and gives background information in the context of this paper. A brief overview of well-known synthesis methods for robust asynchronous logic realization, based on the delay-insensitive data encoding convention (mostly dual-rail) is provided in Section 3. Definitions of some terminologies are given in Section 4. The proposed heuristic to derive minimum disjoint sum-of-products expression from a reduced sum-of-products expression is elucidated in Section 5, and its efficiency vis-à-vis other methods is studied on the basis of some combinational benchmarks. The novel heuristic to obtain minimum orthogonal sum-of-products format corresponding to an arbitrary combinational logic specification, on the basis of dual-rail encoding, is discussed in Section 6. Results corresponding to the self-timed realization of many combinatorial benchmarks are also given in this section. The new self-timed system architecture that benefits the above synthesis mechanism is presented in Section 7. Finally, we make the concluding remarks in Section 8. Initial insights regarding extension of the proposed two-level heuristic to multiple levels are also mentioned therein.

2. Self-timed logic block

A self-timed logic block is also referred to as a function block and represents the robust asynchronous equivalent of a traditional synchronous combinational logic circuit. In addition to realizing the requisite functionality, the self-timed logic block has to be transparent to the handshaking, as implemented by its surrounding latches. Besides, there should not be any dangling inputs or outputs within the function block.

A self-timed logic block is entrusted with the responsibility of *indicating* (acknowledging) the completion of the computation on all its intermediate nodes. With respect to the manner of indication, such logic blocks are either classified as strongly indicating or weakly indicating. A depiction of strong and weak-indication timing models, as specified by Seitz [3], is shown in figure 1. In essence, strongly indicating logic blocks wait for the arrival of all the inputs (whether valid or spacer data) before they start to produce any

output (valid or spacer). On the other hand, a weakly indicating logic block is allowed to produce a subset of the outputs based on a subset of the inputs. All inputs must become valid before the last output can become valid; the last output must not become invalid until all of the inputs become empty. It was also shown in [3] that a legal interconnection of strong or weak-indication logic blocks is permissible.

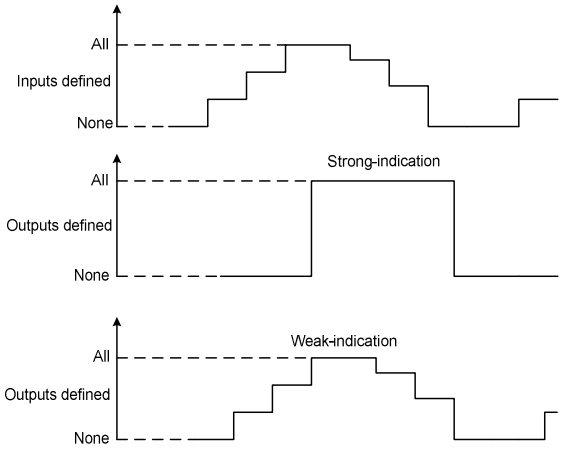

Figure 1: Portraying strong and weak-indication

Delay-insensitive data encoding and return-to-zero handshaking (4-phase handshaking) constitutes a widely preferred robust signaling convention for self-timed designs. With dual-rail encoding, a data wire d is represented using two rails: d^0 and d^1. A transition on the former indicates the transmission of a *zero*, while a transition on the latter indicates the transmission of a *one*. The condition when both d^0 and d^1 are zeroes signifies the spacer (empty) state and all the data wires assume this state between two valid data states. d^0 and d^1 are not allowed to be high simultaneously, as the coding scheme is unordered. The request signal from the sender is embedded within the encoded data wires.

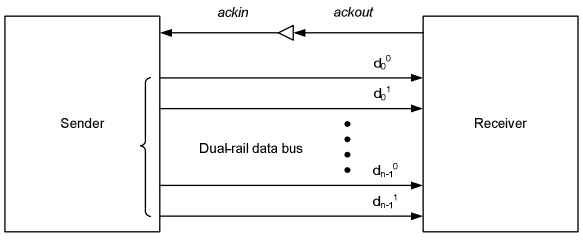

Figure 2: Depiction of delay-insensitive data encoding and 4-phase handshaking

The return-to-zero handshake protocol is explained through the following steps:

i. The dual-rail data bus is initially in the spacer state. The sender transmits the codeword (valid data). This results in 'low' to 'high' transitions on the bus wires (i.e. any one of the rails of all the dual-rail signals is assigned a logic 'high' state), which correspond to non-zero bits of the codeword.

ii. After the receiver receives the codeword, it drives the *ackout* (*ackin*) wire 'high' ('low').

iii. The sender waits for the *ackin* to go 'low' and then resets the data bus (i.e. it is driven to the spacer state).

iv. After an unbounded, but finite (positive) amount of time, the receiver drives the *ackout* (*ackin*) wire 'low' ('high'). A single transaction is now said to be complete and the system is ready to proceed with the next transaction.

3. Review of self-timed design methods

The unorthodox methods employed to synthesize self-timed combinatorial logic usually incur substantial area overhead and are beset with the problem of input space explosion, which poses an exponential complexity of $O(2^n)$ with the number of primary inputs of $O(n)$. Here, n denotes the number of concurrent single-rail inputs. A self-timed logic block realization typically satisfies the acknowledgement property and the unique-successor-set property [4]. The acknowledgement property specifies that every transition on a gate output node, excepting the primary outputs, should be accompanied by a successive transition on another gate output in the subsequent stage. The unique-successor-set property stipulates that the monotonic cover condition [5], which ensures hazard-free implementation of speed-independent (SI) circuits, be incorporated into the description of the logic functionality. In simple terms, the monotonic cover constraint requires that only one product term in a sum-of-products implementation is allowed to assume a logic 'high' at any time in case of either set (true output) or reset (false output) functions.

In general, the cover constraint entails the enumeration of all possible distinct input combinations. Therefore, many indicating logic realization schemes (especially strong-indication methods) suffer from large area overheads and this has usually restricted self-timed logic implementations to those with fewer inputs. Self-timed implementation of larger combinational circuits could incur at least three times the area penalty of a conventional synchronous realization. However, many approaches have been proposed (especially on

the basis of dual-rail encoding) and they differ in the way of dealing with this problem by either:

i. Assuming the presence of the entire state space without mention of any scheme for SI decomposition [3] [6] or
ii. Confining themselves to only full custom solutions for smaller functions [7] or
iii. Circumventing this problem considerably by usually relying upon de-synchronization, with the additional provision of availability of full custom library gates as part of a standard cell library [8] [9] [10] [11] or
iv. Performing SI logic decomposition with/without consideration of the entire input state space [12] [13]

Among these, variants of the third approach have been predominant and facilitate weakly indicating solutions while being technology-dependent. They have their roots in the DIMS approach [6] and the dual-rail combinational logic implementation style [14], which do not synthesize self-timed circuits based on specifications such as communicating handshake processes or signal transition graphs. Instead, they rely upon synchronous CAD tools for initial synthesis and then resort to replacement of every gate with a dual-rail encoded gate pair in a template based fashion that are subsequently mapped using NULL convention logic (NCL) operators. These operators are developed on the basis of threshold logic and are made available as custom elements of a standard cell library.

In contrast, we envisage a new technology-independent multi-level self-timed synthesis technique that is different from any of the NCL based schemes. Towards this end, a preliminary (two-level) synthesis procedure is proposed demonstrating its feasibility for synthesizing combinational functions of any size as self-timed circuits without exploding the input space. In fact, this is cumbersome with approaches (i), (ii) or (iv) mentioned above.

4. Preliminaries

Some definitions are first stated in this section to help with further discussion.

4.1. Definition 1. A *literal* is a symbol referring to a variable (x) or its complement (x'). In case of dual-rail encoding, the notion of a literal is used to refer to either the true-bit ($x1$) or the false-bit ($x0$) representation of a variable (x) respectively.

4.2. Definition 2. A *cube* is defined as a logical product[1] (conjunction) of different literals, where a variable appears in only one of its symbolic notations. For example, *a'b*, *abc'd* are single-rail cubes. With dual-rail encoding, a cube specifies a logical conjunction of the true-bits or false-bits of different variables. $a0b1$ and $a1b1c0d1$ are the respective dual-rail encoded product term equivalents of the single-rail cubes mentioned earlier.

4.3. Definition 3. A Boolean function*, f,* is a mapping of type f: $\{0,1\}^n \rightarrow \{0,1,d\}$, where d denotes a *don't care* condition. If d does not exist, then the function f is said to be completely specified or two-valued, otherwise it is called incompletely specified or ternary. Each of the 2^n nodes in the Boolean space corresponds to a canonical product term (minterm). The ON-set, OFF-set and DC-set of f correspond to those minterms that are mapped to 1, 0 and d respectively.

4.4. Definition 4. A *cover* is a set of irredundant product terms (prime implicants) pertaining to a logic specification and the cardinality of a cover is the number of essential products comprising the cover.

4.5. Definition 5. A sum-of-products (SOP) form consists of a disjunction of product terms, each of which involves a conjunction of literals. If the number of terms in a SOP form is the least possible, then the SOP is referred to as minimum SOP.

5. Disjoint sum-of-products form

A Boolean equation is said to be in disjoint SOP (DSOP) form, if it is described by a logical sum of product terms that are all disjoint [15], i.e. no two product terms cover a common minterm when expanded. A DSOP form with the least number of product terms is known as minimum DSOP form. While SOP minimization can be likened to a set covering problem, DSOP minimization can be likened to the problem of finding a minimum disjoint cover, the exact solution of which is NP-hard [15]. For example, the number of irredundant product terms of the SOP expression of an Achilles' heel function [15] is given by $O(n/2)$, while the number of essential products constituting its DSOP expression is specified by $O(2^{n/2}-1)$, where 'n' represents the number of distinct primary inputs. DSOPs have been traditionally used in

[1] The Muller C-element typically serves as the conjunction operator. It outputs a high (low) when all its inputs are high (low), otherwise it retains its state. It is an AND gate for transitions.

several applications in CAD areas, for example, calculation of spectra of Boolean functions [16] or as a starting point for the minimization of exclusive-OR SOP logic [15]. This in turn forms the backbone of synthesis schemes for reversible logic circuits [17] that have applications related to the field of quantum computing. It has been found that DSOP solutions directly generated by Espresso are generally far from the optimum, especially in case of functions with several inputs because it considers group minimization of all the function outputs. An alternative approach would be to consider deriving DSOP solutions for the combinational function outputs on an individual basis on the basis of their SOP forms using heuristics [18] – [22], where the reduced SOP form of a non-minimized logic function can be obtained through multi-output minimization using a standard two-level logic minimizer: Espresso. A majority of the heuristics proposed earlier, following this strategy, were found to yield better solutions for many case studies in comparison with the DSOP solutions directly generated using Espresso.

5.1. Proposed heuristic

The heuristic proposed to derive DSOP format from a SOP format is explained below:

- *Step 1:* Obtain the SOP form of a logic function.
- *Step 2:* Compare each cube with every other cube in the SOP form to check whether they are mutually disjoint. If and only if each cube does not overlap with every other cube in the MSOP form, then go to Step 11, else proceed with Step 3.
- *Step 3:* Enumerate all the overlapping pairs of cubes that have a non-disjoint support. If only pairs of cubes with disjoint support exist, go to Step 8, else proceed with Step 4.
- *Step 4:* From among the overlapping pairs of cubes that feature a non-disjoint support, choose that pair of cubes which comprises the highest degree of logic sharing among its constituents. If many such pairs of cubes exist, which exhibit a similar highest degree of commonality then an arbitrary choice is resorted to.
- *Step 5:* Use the *distributive axiom* to extract the kernel. Apply the converse of the *absorption axiom* of Boolean algebra to transform the kernel comprising overlapping cubes with disjoint support into non-overlapping cubes with a non-disjoint support. Apply the distributive property of Boolean algebra to re-enumerate the product terms.
- *Step 6:* Check whether any cube contains any other cube in the function; if so, the covering cube is made to absorb the covered cube. Also, check whether any cube is duplicated in the logic function. If so, the redundancy is eliminated by applying the *idempotency axiom*.
- *Step 7:* Go to Step 2.
- *Step 8:* Consider any two cubes with a disjoint support, which also have the least support set cardinalities. If many choices result, then a random selection is made. Between such a pair of cubes, the *identity axiom* of Boolean algebra is applied to any of the pair of cubes considered. This results in a cube expansion by making use of the distributive axiom.
- *Step 9:* If any cube is found to cover any other existing cube in the function, the covered cube is discarded and the covering cube is alone retained. Logic duplication is eliminated using the idempotent law of Boolean algebra.
- *Step 10:* Return to Step 2.
- *Step 11:* Terminate the routine as the desired DSOP solution has been obtained.

The logical correctness of the resulting DSOP solution is guaranteed by the Boolean axioms used, which are well-established and proven properties. The functional correctness of the DSOP solution is ensured by comparison of each cube with every other cube forming the cover of each function output to make sure that they do not overlap. The combinational equivalence of a SOP form and its corresponding DSOP form is confirmed through the 'Dverify' option of Espresso. The cost of the DSOP solution is represented by the count of all the unique input cubes, some/all of which may eventually be found to be shared between the various outputs.

5.2. Illustration and results

The DSOP heuristic described above has been implemented in Java and has been used to generate results for some combinatorial benchmarks specified in PLA format. Minimum SOP and DSOP forms of the benchmark, *newtag*, obtained using Espresso, are represented by means of the cube-variable matrices of figures 3 and 4 respectively for illustration purpose. The benchmark function has a single output and its support set is composed of 8 elements. The cube-variable matrix is an O($m \times n$) matrix, where 'm' specifies the number of irredundant cubes of the function (rows of the matrix) and 'n' refers to the number of unique input support variables of the function (columns of the matrix). A '1' entry at the intersection of a particular row and column index (a_{mn}) implies the existence of a variable in its normal form, while '0' and '-' entries signify the inverted and don't care states of the variable respectively. The conjunction

of all the variables in a row, appearing in either their normal or complementary forms, describes the cube corresponding to that row of the matrix. The logic function is expressed as $F = \sum C_i$, where $i = 1,\ldots,m$; i.e. the summation of m non-redundant cubes that may have a maximum dimension of n. In a DSOP cube-variable matrix, $a_{pq} \neq a_{rq}$, for any pair (p, r) of m with respect to at least a column q of the matrix, where $p \neq r$ and q signifying a column index.

	a	b	c	d	e	f	g	h
df'h'	–	–	–	1	–	0	–	0
de'h'	–	–	–	1	0	–	–	0
df'g'	–	–	–	1	–	0	0	–
de'g'	–	–	–	1	0	–	0	–
de'f'	–	–	–	1	0	0	–	–
c	–	–	1	–	–	–	–	–
b'	–	0	–	–	–	–	–	–
a	1	–	–	–	–	–	–	–

Figure 3: Cube-variable matrix of SOP form of *newtag*, based on Espresso

a	b	c	d	e	f	g	h
–	0	–	–	–	–	–	–
–	1	–	1	–	0	0	–
–	1	–	1	0	1	0	–
–	1	–	1	–	0	1	0
–	1	–	1	0	1	1	0
–	1	–	1	0	0	1	1
–	1	1	0	–	–	–	–
–	1	1	1	1	1	–	–
–	1	1	1	0	1	1	1
–	1	1	1	1	0	1	1
1	1	0	0	–	–	–	–
1	1	0	1	1	1	–	–
1	1	0	1	0	1	1	1
1	1	0	1	1	0	1	1

Figure 4: Cube-variable matrix of DSOP form of *newtag*, based on Espresso

Figure 5 depicts the cube-variable matrix of the DSOP form of the benchmark *newtag* corresponding to our proposed heuristic. It can be inferred from this example that the cost (number of irredundant cubes) of our DSOP heuristic is similar to the cost of the SOP solution of Espresso. Thus, our DSOP procedure has effected reduction in the number of essential non-overlapping product terms by 43% when compared with the DSOP solution of Espresso.

a	b	c	d	e	f	g	h
0	1	0	1	1	0	1	0
0	1	0	1	0	1	1	0
0	1	0	1	1	0	0	–
0	1	0	1	0	1	0	–
0	1	0	1	0	0	–	–
0	1	1	–	–	–	–	–
0	0	–	–	–	–	–	–
1	–	–	–	–	–	–	–

Figure 5: Cube-variable matrix of DSOP form of *newtag*, corresponding to the proposed heuristic

A set of combinational functions of the MCNC benchmark suite were considered to evaluate the potential of the proposed heuristic vis-à-vis the DSOP solutions rendered by other methods. These are shown in Table 1. The 'dash' in certain positions of the Table indicates the unavailability of a result for the benchmark corresponding to a specific method in the literature. The optimal solution for a benchmark based on a particular method(s) is highlighted in 'bold-face'.

From Table 1, it can be observed that the proposed heuristic has facilitated optimal/near-optimal solutions for a majority of problems. Among the logic functions, *alu4*, *cordic*, *max1024* and *x7dn* are relatively bigger specifications, with *alu4*, *max1024* and *x7dn* having been classified as hard problems in the original Espresso benchmark suite. In comparison with the DSOP solution rendered by Espresso, the proposed method reports a substantial reduction in the number of essential products by 65%. Compared to the SOP expression generated using Espresso, the proposed DSOP heuristic based solution is found to be greater by 3.3×, while the DSOP solution of Espresso is found to be more expensive than its logically equivalent SOP format by 9.5×. With respect to the bigger specifications, the proposed method enables a reduction in the number of essential DSOP terms by 15.5% compared to the best solution rendered by the other heuristics. When considering all the benchmarks, the proposed heuristic facilitates a cost reduction of the order of 14.3%, on a mean basis.

Table 1: Cost of DSOP form of some combinatorial benchmarks corresponding to different heuristics

Benchmark	Espresso	[17]	[18]	[19]	[20]	[21]	Proposed
5xp1	62	70	-	-	82	79	**48**
alu4	3551	-	-	-	1545	1372	**1206**
b12	654	**57**	-	-	60	60	62
clip	359	**162**	-	-	262	212	167
cordic	22228	-	-	-	19763	8311	**6687**
max1024	776	-	-	-	444	-	**362**
misex1	18	**15**	-	-	34	34	**15**
misex2	29	**28**	-	-	30	29	**28**
mlp4	206	-	-	-	203	-	**155**
rd53	**31**	**31**	-	-	35	35	**31**
rd73	**127**	**127**	-	-	147	147	**127**
rd84	**255**	-	-	-	294	294	**255**
x7dn	1697	-	-	-	**1091**	-	1228
xor5	**16**	-	**16**	**16**	**16**	**16**	**16**
Z9sym	190	-	186	**148**	-	-	171

6. Orthogonal sum-of-products form

The orthogonal sum-of-products (OSOP) form [14] corresponding to a self-timed logic block output consists of product terms that are all orthogonal to each other, i.e. the cubes do not overlap. Every cube is orthogonal to every other cube in an OSOP expression and therefore it would inherently satisfy the monotonic cover constraint. While the DSOP form [15] is also composed of non-overlapping product terms, the OSOP form is distinguished in that it comprises encoded outputs described using encoded inputs. The OSOP form with the least number of product terms can be referred to as minimum OSOP form.

6.1. Proposed synthesis strategy

The problem of deducing the efficient OSOP form for a dual-rail encoded function block (inclusive of both its 'true' and 'false' outputs) can be narrowed down to finding a minimum DSOP solution for the outputs and their complements of a combinational logic specification. Here, the function block represents the dual-rail encoded equivalent of an original synchronous combinational logic description. This is possible because the output and its complement of a combinational circuit can be identified as the true and false outputs of a dual-rail encoded self-timed equivalent with a straightforward literal replacement, as is the case with DIMS [6] or NCL based approaches [8] – [11]. Thus the OSOP synthesis scheme relies upon the DSOP synthesis procedure as its back-end.

In view of this, it can be concluded that the efficacy of the DSOP heuristic could have a direct bearing on the effectiveness of the resulting OSOP solutions for a self-timed logic block construction. The ON-sets and DC-sets of the logic function outputs and their corresponding OFF-sets are considered separately so as to obtain the SOP forms for the combinational function outputs and their complements. The corresponding DSOP expressions are then derived for the function outputs and their complements in parallel. However, the DSOP procedure purely corresponds to synchronous logic. Therefore, the OSOP form of a self-timed logic block is obtained by invoking the DSOP heuristic for both the ON and OFF-sets of a combinatorial function specification simultaneously, followed by subsequent encoding (here, dual-rail encoding) of the inputs and outputs. The OSOP heuristic has also been implemented in Java on the basis of the DSOP synthesis method.

6.2. Results

Asynchronous dual-rail equivalents of a number of combinatorial benchmark functions were considered to estimate the cost of their OSOP forms. The OSOP cost shown in column 4 of Table 2 reflects the number of unique cubes. It can be seen that, on an average, the OSOP heuristic has resulted in solutions, which encompass approximately 21% logic (cubes) sharing, but could be higher in specific cases. For example, in case of *ex5*, an 83% reduction in the number of cubes has been achieved through sharing of common logic.

Table 2: Cost of self-timed realization of combinational benchmark functions

Function block	# Inputs in dual-rail format	# Outputs in dual-rail format	# Orthogonal product terms		Runtime (Minutes: Seconds)
			After sharing	Before sharing	
9sym	18	2	251	251	0:2
al2	32	94	310	472	0:0
alu4	**28**	**16**	**2711**	**2803**	**3:16**
amd	28	48	358	613	0:0
apex3	108	100	1351	2504	0:9
b3	64	40	1298	1507	1:7
bcd	52	76	6831	7942	10:15
chkn	58	14	523	526	0:17
cps	48	218	3390	5001	0:36
duke2	44	58	705	955	0:1
e64	130	130	2376	3033	3:55
ex4	**256**	**56**	**1062**	**1062**	**0:5**
ex5	16	126	346	2054	0:2
exep	60	126	3177	3591	0:27
ibm	**96**	**34**	**1365**	**1366**	**0:14**
in3	70	58	496	813	0:1
in4	64	40	1396	1673	1:11
intb	60	28	2320	2320	3:10
jbp	**72**	**114**	**636**	**869**	**0:1**
luc	16	54	175	496	0:0
max1024	20	12	663	826	0:6
misex3	28	28	3826	4422	3:5
misg	**112**	**46**	**189**	**192**	**0:0**
mish	**188**	**86**	**163**	**173**	**0:0**
misj	**70**	**28**	**58**	**69**	**0:0**
mlp4	16	16	319	403	0:0
newapla	24	20	82	94	0:0
newill	16	2	19	19	0:0
opa	34	138	572	1464	0:2
pdc	**32**	**80**	**1358**	**1740**	**0:4**
ryy6	32	2	155	155	0:4
sao2	20	8	202	258	0:0
shift	**38**	**32**	**200**	**212**	**0:0**
soar	**166**	**188**	**1269**	**1566**	**0:3**
spla	32	92	1577	2209	0:7
sym10	20	2	478	478	0:24
t1	42	46	384	501	0:0
t481	32	2	2142	2142	26:23
ti	**94**	**144**	**1388**	**2500**	**0:5**
ts10	**44**	**32**	**272**	**512**	**0:0**
vg2	50	16	632	647	0:6
x6dn	78	10	432	603	0:5
x7dn	**132**	**30**	**3711**	**3752**	**2:1**
Z9sym	18	2	243	243	0:1

The benchmarks highlighted in 'bold-face' in Table 2 correspond to dual-rail asynchronous equivalents of hard combinational logic specifications. The synthesis time corresponds to the heuristic running on an Intel Core2 Duo processor at 2.4GHz under Windows XP with a 1GB RAM. In case of self-timed logic blocks that comprise only two encoded outputs, for example, *9sym*, *newill*, *ryy6*, *sym10*, *t481* and *Z9sym*, it should be obvious that hardly any logic sharing is feasible. The synthesis of function blocks *9sym*, *newill*, *ryy6*, *sym10*, *t481* and *Z9sym* reveals that the package's processing time depends on the number of inputs and the logic description. With respect to *soar*, [3] [6] and [13] would consider the entire input space, which is of $O(2^{83})$. In contrast, the OSOP heuristic results in only 1269 product terms. It is to be noted here, that the recent two-level weak-indication synthesis method [23] is unlikely to cope with larger combinatorial problems. This is owing to its selective expansion of products to achieve distributive indication; hence its scalability is dependent on the block size.

7. Proposed self-timed system topology

Our self-timed logic block realization method has only ensured that the cover constraint is satisfied without addressing the system indication aspect. The architecture that externally takes care of the indication phenomenon is shown in figure 6. It generally favors a weakly indicating realization by synchronizing only a pair (true & false) of encoded outputs with the current stage completion detection logic using the synchronizer, before being fed to the next stage.

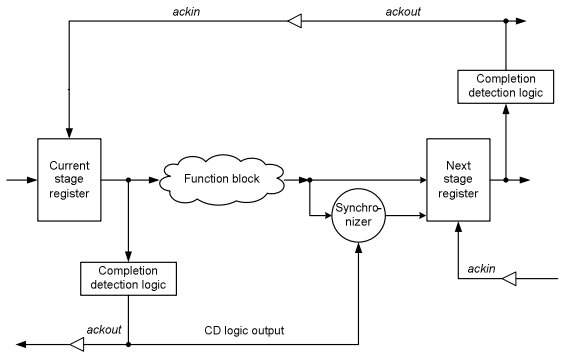

Figure 6: Proposed robust system configuration

8. Conclusion and scope for further work

A two-level synthesis method for implementing combinatorial logic as compact self-timed circuits is presented in this paper. For multi-level synthesis, 'complementary cubes insertion' is proposed as a supplementary concept. It necessitates the insertion of complementary OFF-set/ON-set cubes in a rail of the function block output, whose complementary rail contains an indecomposable ON-set/OFF-set term respectively. Implementation of this procedure and analyzing various delay-insensitive data encodings within this framework are left for future work.

9. References

[1] SIA's ITRS report 2008, Available: http://www.itrs.net

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

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

[4] A.J. Martin, "The limitation to delay-insensitivity in asynchronous circuits," *Proc. ARVLSI*, pp. 263-278, 1990.

[5] A. Kondratyev et al., "Basic gate implementation of speed-independent circuits," *Proc. DAC*, 1994, pp. 56-62.

[6] J. Sparso et al., "Delay-insensitive multi-ring structures," *Integration, the VLSI Journal*, vol. 15, 1993.

[7] T.E. Williams, "Self-timed rings and their application to division," *PhD dissertation*, Stanford Univ., May 1991.

[8] M. Ligthart et al., "Asynchronous design using commercial HDL synthesis tools," *ASYNC*, 2000, 114-125.

[9] A. Kondratyev et al., "Design of asynchronous circuits by synchronous CAD tools," *IEEE D&T*, vol.19, 2002.

[10] Y. Zhou et al., "Cost-aware synthesis of asynchronous circuits based on partial acknowledgement," *ICCAD*, 2006.

[11] C. Jeong et al., "Optimization of robust asynchronous circuits by local input completeness relaxation," *ASP-DAC*.

[12] B. Folco et al., "Technology mapping for area optimized QDI circuits," *Proc. IFIP VLSI SoC*, 2005.

[13] W.B. Toms et al., "Efficient synthesis of SI combinational logic circuits," *ASP-DAC*, 2005, 1022-1026.

[14] V.I. Varshavsky (Ed.), *Self-timed Control of Concurrent Processes*, Kluwer Academic Publishers, 1990.

[15] T. Sasao (Ed.), *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, Dordrecht, 1999.

[16] M.A. Thornton et al., *Spectral Techniques in VLSI CAD*, Kluwer Academic Publishers, Boston, MA, 2001.

[17] P. Gupta et al., "An algorithm for synthesis of reversible logic circuits," *IEEE TCAD*, 25 (11), Nov. 2006.

[18] B.J. Falkowski et al., "An effective computer algorithm for the calculation of disjoint cube representations of Boolean functions," *36th MWSCAS*, 1993, pp. 1308-1311.

[19] T. Kozlowski, "Application of exclusive-OR logic in technology independent logic optimization," *PhD Th.*, 1996.

[20] L. Shivakumaraiah et al., "Computation of disjoint cube representations using a maximal binate variable heuristic," *Proc. 34th IEEE SSST*, 2002, pp. 417-421.

[21] G. Fey et al., "Utilizing BDDs for disjoint SOP minimization," *45th MWSCAS*, 2002, pp. II-306-II-309.

[22] N. Drechsler et al., "Disjoint SOP minimization by evolutionary algorithms," *EvoWorkshops*, 2004, 198-207.

[23] W.B. Toms et al., "Prime Indicants: a synthesis method for indicating combinational logic blocks," *ASYNC,* 2009.