Asynchronous Design Using Commercial HDL Synthesis Tools.

Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, Alex Kondratyev

Theseus Logic, 710 Lakeway Dr., Suite 230, Sunnyvale, CA, 94086, USA

{michiel.ligthart, karl.fant, ross.smith, alexander.taubin, alex.kondratyev}@theseus.com

Abstract

New design technologies rely on truly reusable IP blocks with simple means of assembly. Asynchronous methodologies could be a promising option to implement these requirements. Promotion of asynchronous design strongly depends upon the "level of service" delivered to the designer. Current asynchronous design tools require a significant re-education of designers and their capabilities are far behind synchronous commercial tools. One solution to these problems, which we advance in this paper, is to stick to a conventional design flow as closely as possible and to use commercial design tools as much as possible. The paper considers a particular subclass of asynchronous circuits (Null Convention Logic or NCL) and suggests a design flow which is completely based on commercial CAD tools. It argues about the trade-off between the simplicity of design flow and the quality of obtained implementations.

1. Introduction

There are two common beliefs about asynchronous design: 1) asynchronous circuits are much more difficult to design than synchronous ones and 2) existing CAD tools for synchronous circuits are of no help for asynchronous design [1,2].

The main aim of this paper is to argue against these statements. The paper considers a particular subclass of asynchronous circuits (Null Convention Logic (NCL) [17,18]) and suggests a design flow that is completely based on commercial CAD tools.

The trend towards Systems-On-Chip (SOC) technology is now widely promoted in electronic industry. The prime goal of SOC technology is to create truly reusable IP blocks that can be quickly built and are guaranteed to work the first time [3]. Together with these high quality IP pieces an overall design methodology should provide a simple means for IP assembly, based on plug 'n' play principles. Being clock free, asynchronous circuits constitute an attractive SOC approach. From the system architecture point of view it is much easier to build using asynchronous rather than synchronous blocks. Having asynchronous IPs would be an important step in developing SOC technology.

Whether the electronic industry would adopt an asynchronous methodology depends on investments (to transfer to a new design flow) and the maturity of CAD support. Though recently a lot of progress has been achieved in the development of asynchronous CAD tools [4,5,15,6], the current tools suffer from two shortcomings: 1) they require a significant re-education of designers (investment problem) and 2) their capabilities are far behind commercial tools from the synchronous domain (CAD support problem). One solution to these problems, which we advance in this paper, is to stick to a conventional design flow as closely as possible and to use commercial design tools as much as possible.

If successful this methodology will be able to provide the following benefits:

- Reduced time to market (HDL automated design, design reuse),
- Plug 'n' play without clock system coordination,
- Low power and low EMI – "by construction"[7, 8],
- Delay-insensitivity (DI) as a way to handle wire delays in deep submicron (DSM) technology.

This paper focuses on performing **logic synthesis** with the help of conventional CAD tools. Logic synthesis algorithms by themselves are not specifically targeted to synchronous design. Clocking is a common and simple way to abstract from timing issues in the behavior of real circuits. Generally speaking, clocking allows a designer to ignore timing issues when considering system functions and describe both the functions performed and the circuits themselves in terms of Boolean algebra (according to Shannon's concept) [9, 10]. As a result clocking pulses and gates' switching in a circuit are causally unrelated and synchronization is done by matching their delays. This leads several well-known difficulties in construction of clocks for big systems [2,11].

By contrast, asynchronous systems coordinate their behaviors in a purely causal way. As a result logic synthesis for asynchronous circuits takes care not only of functionality but also of the proper ordering of the gates' switching. This makes logic synthesis for conventional asynchronous circuits much more complicated than for synchronous circuits [12,13,15,19,23], which results in a strong dislike of the synchronous community towards asynchronous methodology. Incorporating asynchronous design into synchronous design flow requires abstracting from timing issues in a way that will not divert logic design methods from the synchronous world yet allow us "to live" without clocks.

Our paper attempts to show that this flow could be implemented by using NCL as a basis for asynchronous implementation. The main questions that the paper answers are:

- Is it possible to use RTL synthesis tools for NCL asynchronous design and generate correct designs?
- What are the results of design using the NCL tools?

The paper is organized as follows. Section 2 introduces the main theoretical concepts. Section 3 places this paper in context among existing works. Section 4 contains an overview of HDL-design flow, which is illustrated by a "toy" design in Section 5. The validity of the suggested approach is proven in Section 6. Experimental results are presented in Section 7.

2. Theoretical background

2.1. Boolean networks

Boolean networks provide a formal representation of combinational circuits for logic synthesis. A Boolean network is a directed acyclic graph. A gate in some combinational circuit is represented by node i in the graph, with name a_i and a completely specified logic function f_i [14]. A directed arc from node i to node j means that variable a_i is explicitly used in the representation of f_j, i.e. a_i is in a **support** of f_j. Direct predecessors of a node a_i are called a_i's **fan-in**, while direct successors of a_i are called a_i's **fan-out**. Some of the nodes in the graph are designated as outputs (inputs) of the network, called **"primary outputs" ("primary inputs").**

Combinations of primary inputs that never occur during operation of a Boolean network contribute to the so-called **external don't care** set of a network. Two networks with the same set of primary inputs and primary outputs are called equivalent if for all values of corresponding primary inputs not in the external don't care sets, the corresponding primary outputs are equal [14]. A Boolean network N is **prime and irredundant** if removing a single literal or cube from any function f_i of its node produces a network N' which is non-equivalent to N.

2.2. Delay-insensitive combinational circuits

A gate a_i of a combinational circuit can be in two states: stable – when the value on its output corresponds to the value of Boolean function f_i computed by the gate's fan-in, and enabled – otherwise. An enabled gate can either switch (fire) after the elapsed gate delay or can return to a stable state because of its fan-in changes. The last case of resolving the enabling is undesirable in asynchronous circuits because it might produce glitches on gate outputs, known in the literature as hazards [10]. A circuit in which hazards never occur under any distribution of delays in wires and gates is called delay-insensitive. An acknowledgement notion plays the key role in ensuring delay-insensitivity.

Informally, we say that the firing of gate a_i acknowledges the firing of gate a_j if the fact that a_i switches after a_j has been enabled indicates that a_j has already switched as well. In a delay-insensitive circuit all the gates in the fan-out of a_i must acknowledge every firing of gate a_i. This guarantees that independently of the skew of wire delays after a_i output (after fork), the information about a_i firing is properly delivered to destination nodes in a network.

Sometimes the requirement of delay-insensitivity (DI) with respect to **every** wire fork in a circuit is overly restrictive for a designer. Some forks could be considered safe because the skew of their wire delays is guaranteed to be less than the minimum gate delay. These forks are called **isochronic** [15]. For isochronic forks it is sufficient to get an acknowledgement from at least one of the gates on the fork fan-out.

2.3. Null Convention Logic

Regular ways of DI circuits implementations are often rely on application of DI codes [16]. These have the attractive property that a receiver is able to determine that a codeword has arrived by a codeword itself, without appealing to timing assumptions.

Null Convention Logic [17,18] is a specific way of implementing data communication based on delay-insensitive encoding. It assumes a two-phase discipline in which data communication alternates between a set and reset phases [19]. Data changes from spacer (called NULL) to proper codeword (DATA) in the set phase and then back to NULL in the reset phase.

In NCL this behavior is pushed down to the level of each particular gate of a circuit. If the current state of a gate is NULL, then the gate keeps its output in NULL until NULL is present in at least one of its fan-ins. Then, when **all** gate fan-ins receive a codeword (DATA), the output of the gate changes to DATA. A gate has a symmetric behavior in the reset phase – it keeps output in DATA until **all** the fan-ins receive NULL; after that the output also changes to NULL. Such gates are called **strongly indicating** [26] because the firing of gate's output acknowledges firing of every fan-in.

This behavior is naturally expressed in a multi-value logic. Let a signal in a Boolean network takes three logic values: T and F for data items 1 and 0 respectively and N for NULL value. Then the behavior of basic gates in NCL logic is described like in Figure 1(a) (gates are assumed to be initially in a state NULL).

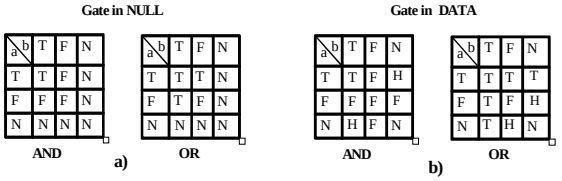

Figure 1. Symbolic tables for basic NCL gates

From the above explanation it follows that NCL gates have sequential behavior because they switch differently depending on the current value on the output. The description of behavior of basic NCL gates is accomplished by symbolic tables for initial state DATA (see Figure 1(b), where H stands for holding the previous DATA state of the gate (T or F) while one of the inputs changes to NULL).

The representation of NCL gates in a three-level logic is called 3NCL [17]. 3NCL logic is a convenient mathematical abstraction but it has no efficient physical implementation due to the binary nature of signals used in design practice.

For physical implementation each signal a in 3NCL is represented by two wires $a.t$ and $a.f$ in a circuit under the following encoding of 3NCL symbolic values:

$$a = T \Leftrightarrow a.t = 1, a.f = 0;$$

$$a = F \Leftrightarrow a.t = 0, a.f = 1;$$

$$a = N \Leftrightarrow a.t = 0, a.f = 0$$

The combination of values $a.t=a.f=1$ is not used. This encoding is known as a dual-rail encoding [20] and it gives one of the simplest DI codes.

Implementation of 3NCL logic through a dual-rail encoding (called 2NCL [17]) gives a physical representation of NCL logic. The sequential behavior of gates in 2NCL is ensured by a feedback from gates' outputs to their fan-ins, which allows us to represent gate's behavior by a logic equation $g=S+gR^{\wedge}$, where S and R are the set and reset (respectively) functions of a gate. A general view on semi-static CMOS implementation of a gate in 2NCL is shown in Figure 2(a).

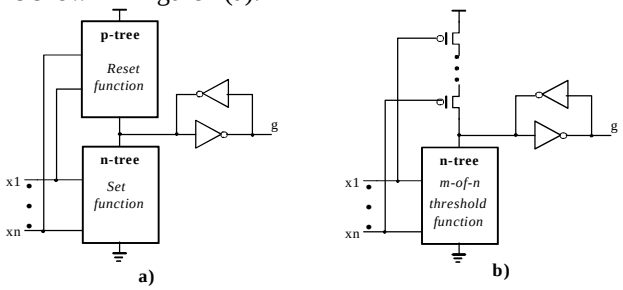

Figure 2. Implementation of NCL gate in CMOS

A refined picture of the gates' structure could be obtained through consideration of specific properties of dual-rail circuits under two-phase (set and reset) operation. These properties are:

1)in a dual-rail circuit a transition from NULL to DATA is monotonic

2)the transition of primary inputs of a combinational circuit from DATA to NULL will set all gates in a circuit into the NULL state

From (1) it follows that a set function S of a gate must be **positively unate** [14], i.e. every variable is met in function S without inversion. In fact set conditions for NCL gates are convenient to specify by a particular subclass of unate functions – **threshold** functions [21].

A threshold function S is the one that can be defined by a system of inequalities: $S(x_1,\ldots,x_n) = 1$ iff $w_1x_1 + w_2x_2 + \ldots + w_nx_n \geq m$, where w_i are the weights, m is the threshold value and "+" is an arithmetic sum. When all weights are 1 a threshold function can be characterized by two numbers n – number of variables, and m – the threshold value. This representation is called an **m-of-n threshold** function. Any positive unate function could be presented as disjunctive or cascaded superposition of **m-of-n threshold** functions [22].

The ability to reset an NCL gate can be easily concluded from (2). An NCL gate changes its output to NULL when all its inputs are NULL. Since DATA values are encoded by "01" or "10" we arrive at $R^{\wedge}(x_1,\ldots,x_n) = x_1 \vee x_2 \vee \ldots \vee x_n$

A refined view on implementation of 2NCL gate is shown in Figure 2(b). Clearly every 2NCL n-input gate has the same reset function that does not depend upon the particular type of a threshold function the

gate implements. This property is crucially important for optimization because the reset behavior of NCL network does not depend upon the transformations which are performed on it. In the rest of the paper we refer to this implementation as **threshold gate with hysteresis**.

3. Comparison to prior Work

This Section gives a short summary on the place of the suggested approach among 1) other methods of synthesis of delay-insensitive circuits and 2) attempts of using conventional HDL languages (VHDL) for design of asynchronous circuits.

3.1. Place of NCL among DI design styles.

The idea of using two-phase discipline in data communication is known for long time. In [23] Chuck Seitz suggested the so-called "weak conditions" which formalized the correctness of a system operation under the two-phase discipline. Implementation of two-phase operation by dual-rail circuits is used elsewhere [15,23,19,24] to name but a few.

A regular method for implementation of an arbitrary Boolean network under a two-phase discipline and dual-rail encoding gives Delay Insensitive Minterm Synthesis (DIMS) [25]. This technique is similar to NCL though it uses a very limited set of threshold gates – C-elements and OR-gates. Thus, there is room for optimization and DIMS implementations are significantly larger than similar designs in NCL. A straightforward generalization of DIMS method that merges two-stage DIMS blocks (C-elements + OR-gates) into a single CMOS gate [26] has better area parameters but still under-exploits optimization possibilities because of the limited basis.

An efficient procedure of DI synthesis starting from high-level behavior specification (CHP) was suggested by Alain Martin in [15]. A CHP description in terms of system events is automatically translated into production rules that describe the set and reset functions for each event. A pair of production rules for an event (set and reset) is implemented as a single CMOS gate. These gates are more general than the threshold gates with hysteresis used in NCL because they have no restrictions on the structure of PMOS network inside the gates. This flexibility results in more possibilities for optimization and thus smaller circuits. The DI properties of implementation are guaranteed by using involved and specific synthesis methods with an intensive peephole optimization. Thus its synthesis approach cannot use synchronous design methods or tools.

An interesting attempt to incorporate asynchronous design into a synchronous design flow was done in phased logic [27]. [27] suggests an effective way of mapping the topology of synchronous circuit into a network of phased logic gates. Phased logic gates use Level-Encoded two-phase Dual-Rail [28] signals and replace clock signals by phasing activity that ferries data values from gate to gate. However, phased logic requires logic synthesis to handle event ordering. To coordinate the order of gate firing, additional signals and gates must be inserted in a circuit to ensure the "safeness" and "liveness" properties of the circuit. This non-trivial procedure does not fit the synchronous design flow.

3.2. Use of conventional HDLs in asynchronous design

The use of conventional HDLs (VHDL/Verilog) as front-end specifications for asynchronous circuits gives two immediate advantages: 1) designers outside the asynchronous community could understand and write these specifications with less effort 2) commercial HDL simulators can be used.

Most of the previous works on using conventional HDLs for asynchronous design targeted these two advantages [29,30,31]. In these papers, VHDL/Verilog is the highest level of specification, and the major part of the methodology relies on specific design procedures and exotic (for synchronous designers) models.

One way of using synthesis facilities from HDLs separates a system into control and datapath [32]. For gluing together the control and datapath, the synthesizable subset of HDL is extended by the notion of channels to implement handshake mechanisms. After adding channels to an HDL description a datapath could be synthesized by conventional RTL-synthesis tools in a micropipeline fashion. For implementation of control [32] uses asynchronous synthesis tools (*petrify* [5] e.g.).

Separate implementation of control and datapath requires a careful timing analysis. This is a significant complication. In the NCL design flow a control part is implemented according FSM-based approach of commercial RTL-synthesis tools (where asynchronous registers are instantiated instead of synchronous registers).

4. NCL-shell: Overview of HDL-design flow.

Targeting the usage of conventional CAD tools RTL descriptions of asynchronous designs should

closely match common synchronous description styles. We are using synthesizable VHDL both for simulation and for synthesis, unlike other approaches which restrict or modify the HDL.

4.1. EDA flow

The NCL design flow uses off-the-shelf simulation and synthesis components (See Figure 3).

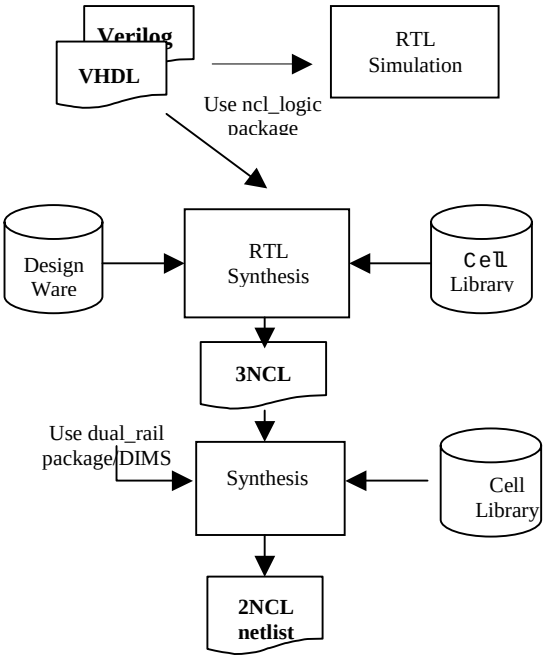

Figure 3. RTL flow for NCL

The flow executes two synthesis steps:

1. **Translate HDL into 3NCL netlist**

The first stage starts with RTL code written with 3NCL, a single-rail multi-valued representation of NCL. For simulation, the NULL value enables the proper set/reset behavior and is displayed as a third state by Model Technology's Model*Sim* VHDL simulator [33]. For RTL synthesis, Design Compiler [35] treats 'N' as a don't care value [34]. This enables the tool to use Boolean synthesis, because it treats 3NCL variables as a single wire. The synthesis tool performs HDL optimizations and outputs an unmapped VHDL dataflow description expressed by AND and INV assignments. This dataflow description is referred to as a 3NCL netlist.

2. Optimize 3NCL into 2NCL netlist

The second stage expands the intermediate 3NCL netlist into a fully dual-rail 2NCL by overloading all AND and INV assignments as DIMS-type dual-rail assignments. This expansion is described in a VHDL package. Stage 2 also performs regular ASIC-type optimization (multilevel minimization of Boolean network [14]), targeting an NCL library.

This flow has been successfully implemented with Design Compiler from Synopsys [35], with Leonardo from Exemplar Logic[36], and with Ambit Envisia [37] from Cadence. A detailed explanation of the above design flow is presented in Section 5 via a small synthesis example.

4.2. NCL coding style

NCL is coded at the register-transfer level (RTL). Unlike the behavioral level, where the synthesis tool determines the placement of registers, the designer must specify the placement of registers in RTL code. This is done either by inference (i.e., writing code following specific rules that the synthesis tool interprets as a register) or by instantiation (i.e., declaring a register and its connections explicitly). Most clocked designs infer registers. We currently must instantiate the registers and specify the register's request and acknowledge signals.

To synthesize and simulate an NCL circuit at the RTL using commercial tools, the tools must handle the NULL value and hysteresis behavior of threshold gates. This is accomplished by following these rules:

- Separate combinational logic and registers. By registers, we mean gates that have request and acknowledge signals. Like clocked logic, the combinational logic is written as concurrent signal assignments or in processes.
- Instantiate NCL registers and provide a simulation-only model with hysteresis behavior. The simulation model is ignored during synthesis
- Use a hysteresis procedure inside processes to simulate hysteresis, but ignore the procedure during synthesis.

In addition, we introduce an NCL-specific simulation package, ncl_logic which:

- Defines type NCL_LOGIC with values {0,1,N, U, X, Z, -} and
- Overloads VHDL operators to incorporate the NULL.

In pseudo-HDL, the approach looks as follows :

comb: PROCESS (sensitivity list)
BEGIN <boolean computation>

```
-- synthesis off
        <ncl hysteresis function>
      -- synthesis on
END PROCESS comb ;
reg: ncl_register
      <register bindings>
```

This approach has the following advantages:

- No limitations on combinational constructs
- Easy to rewrite clocked HDL designs
- The same HDL description can be used for simulation and synthesis.

The remainder of this paper will focus on the synthesis aspects of the methodology and ignore the details of simulation.

5. NCL design flow example.

This section illustrates the design flow using an **if-then-else** statement that is implemented as a 2-1 multiplexer (MUX). This statement is frequently used in RTL. Its efficient implementation has a significant impact on the size of the final circuit**.**

The RTL VHDL program is shown in Figure 4. In terms of Boolean functions MUX behavior is specified by: $z = s \cdot a + \bar{s} \cdot b$

The 3NCL representation of the MUX function is indistinguishable from the synchronous circuit – its sum-of-products implementation is shown in Figure 5(a) while Figure 5(b) shows the same circuit implemented with NAND gates. Optimization of a larger circuit can be performed at this stage. The output of the 3NCL stage is **always** a Boolean network of **two-input** logic gates (two-input NANDs network in particular). The most important step in understanding the NCL methodology is "what is done when 3NCL is expanded into dual-rail 2NCL".

The expansion is performed automatically by replacement of each two-input gate by its dual-rail DIMS implementation. Mapping of NAND gate into a DIMS functional block is shown in the truth table in **Figure 6** where dual-rail signals *a* and *b* are encoded as follows: *N*=00, *T*=10, *F*=01. This conversion to DIMS produces a DI circuit because each DIMS block is strongly indicating [25,26]. Note that it is essential that every minterm in the function domain be acknowledged by a single C-element; merging minterms (and corresponding C-elements) violates DI properties.

```
 library ncl;
use ncl.ncl_logic.all;
ENTITY ifthen IS PORT ( a, b, s  : IN    ncl_logic;
   z     : OUT ncl_logic);
END ifthen ;

ARCHITECTURE ncl OF ifthen IS
BEGIN
test:
   PROCESS(a, b, s)
   BEGIN
    if (s = '1') then  z <= a;
         else z <= b;
         end if ;
  END PROCESS test;
END ncl ;
```

Figure 4. RTL VHDL specification

Replacement of gates from Figure 5(b) with DIMS blocks (see **Figure 6**) gives a circuit with 12 two-input C-elements and 3 three-input OR gates (see Figure 7)

If this circuit were implemented in semi-static CMOS, it would have 114 transistors versus 14 transistors for a synchronous circuit. The area penalty could be reduced by merging three C-elements and the three-input OR gate into a single CMOS gate that uses 13 transistors. Using three of these gates and three C-elements requires 63 transistors. Below we will show that by using Design Compiler for optimization of DIMS circuit this result can be further improved.

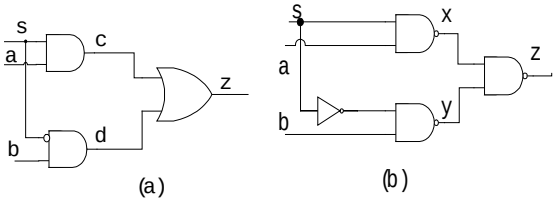

Figure 5. A 2-1 Multiplexer implemented with (a) Sum-of-products and (b) NAND gates.

In order to optimize with Design Compiler (which knows how to work with Boolean logic gates but unaware of threshold gates with hysteresis) the DIMS implementation needs to be mapped into Boolean logic. A direct mapping is not possible because C-elements are sequential. This problem is resolved if the two phases (set and reset) of a circuit operation are handled separately during optimization. In fact for a two-phase protocol the circuit functionality is fully expressed by the **set phase**; the reset phase is performed uniformly for all circuits. The reset phase is guaranteed automatically by the use of threshold gates with hysteresis (see Section 6 for details). Therefore optimizations are applied to the circuit functioning in the set phase. Optimization of DIMS implementation starts by remapping DIMS blocks into "image patterns", which are Boolean combinational gates. DIMS block and the

corresponding image gate have equivalent behaviors in a set phase of circuit operation. The result of remapping is called **smoothing** of a DIMS circuit.

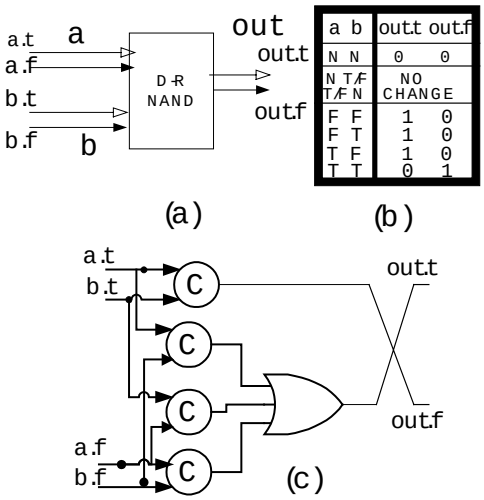

Figure 6. Dual-rail NAND-gate: (a) symbol; (b) truth table; (c) implementation

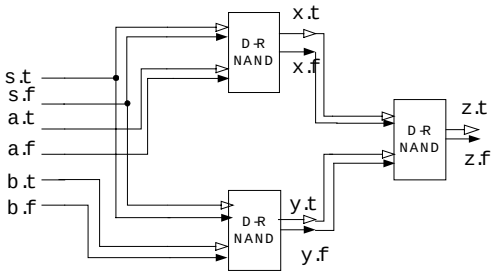

Figure 7. Dual-rail expansion for NAND implementation of MUX

To obtain a smoothing circuit let us consider the functioning of DIMS circuit in a set phase. The output of a 3NCL stage gives a representation of Boolean function through a multi-level combinational circuit of **two-input** logic gates. Thus, the corresponding DIMS blocks contain two-input C-elements and OR gates only. A two-input C-element is described by the Boolean equation: $z = a \cdot b + z(a + b)$. In the beginning of a set phase all gates in DIMS circuit are in the NULL state. Under this initial state ($z = a = b = 0$) an equivalent representation for C-element in a set phase would be an AND gate: $z = a \wedge b$.

Mapping each C-element of a DIMS implementation into AND gates gives the required smoothing circuit (Figure 8 shows a smoothing circuit for NAND DIMS from **Figure 6**(c)). This is the starting point for optimization.

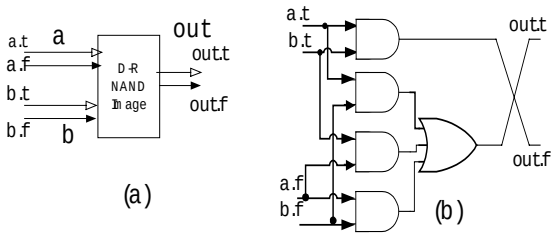

Figure 8. Image of Dual-rail NAND-gate: (a) symbol; (b) functionality.

Design Compiler then proceeds in following steps:

- A smoothing circuit passes technology independent optimization (which is an optimization of a Boolean network representing smoothing circuit)
- Theseus Logic has developed a complete library of four-inputs threshold gates with hysteresis (limitation of four inputs comes from CMOS technology restrictions). This library for each gate contains also its smoothing image: AND gate for C-element, AB+AC+BC gate for 2-of-3 threshold gate with hysteresis, etc. The optimized smoothing circuit is mapped into this library (mapping is done by images)
- The NCL circuit is produced by replacement of each image by its corresponding threshold gate from the library.

if-then-else example continued.

Let us apply the above design flow to the synthesis of the if-then-else statement. Smoothing of the DIMS circuit produces the following result:

$$z.t = x.f \cdot y.f + x.f \cdot y.t + x.t \cdot y.f =$$
$$= a.t \cdot s.t \cdot s.f \cdot b.t + s.t \cdot a.t(s.t \cdot b.f + s.t \cdot b.t +$$
$$+ s.f \cdot b.f) + s.f \cdot b.t(s.f \cdot a.f + s.f \cdot a.t +$$
$$+ s.t \cdot a.f)$$

and

$$z.f = x.t \cdot y.t = (s.f \cdot a.f + s.f \cdot a.t + s.t \cdot a.f) \bullet$$
$$\bullet (s.t \cdot b.f + s.t \cdot b.t + s.f \cdot b.f)$$

Taking into account external don't cares coming from dual rail encoding (for any dual-rail variable *v* the combination $v.t = v.f = 1$ is prohibited) the functions might be simplified to:

$$z.t = a.t \cdot s.t(b.f + b.t) + b.t \cdot s.f(a.f + a.t)$$

and

$$z.f = s.f \cdot b.f(a.f + a.t) + s.t \cdot a.f(b.f + b.t).$$

Substitutions $\alpha = s.f(a.f + a.t)$; $\beta = s.t(b.f + b.t)$; give an optimized network:

$z.f = b.f \bullet \alpha + a.f \bullet \beta$; $z.t = a.t \bullet \beta + b.t \bullet \alpha$;

Each gate in a network is an image of a corresponding threshold gate from the library:

1 . α and β are images (with a pattern "A(B+C)") for the gate "th23w2" (or 2-of-3 threshold gate with weight of input *A* equal to 2),

2 . *z.t* and *z.f* corresponds to the pattern (AB+CD) , which is mapped into "thXOR" gate.

Gate "thXOR" could be implemented through C-elements and an OR gate (in a threshold-disjunctive form (Figure 9 (a))) or by a threshold cascade form (Figure 9 (b)). Its direct implementation by a single CMOS gate is shown in **Figure 10**. The complexity of the NCL 2-1 MUX implementation is 44 transistors which is about 30% better than the optimized DIMS circuit.

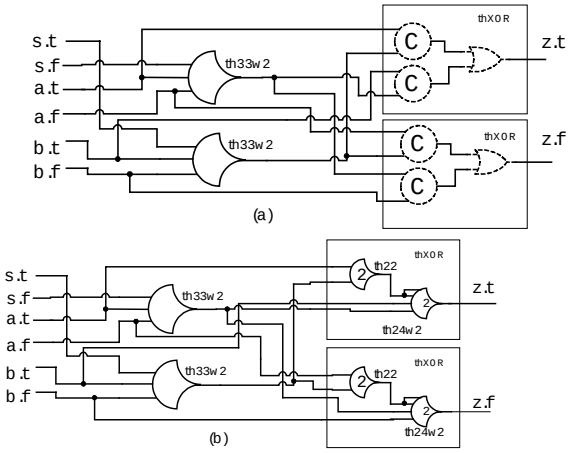

Figure 9 NCL MUX implementation. Threshold-disjunctive (a) and cascade (b) forms

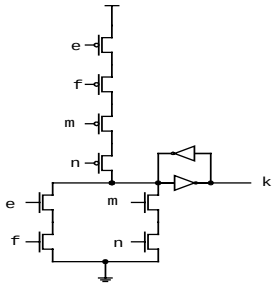

Figure 10. Semi-static CMOS implementation of thXOR.

6. Validation of optimization.

The validity of transformations that are applied to the circuit within NCL synthesis is based on two properties: 1) functional equivalence of optimized and original circuits and 2) strong indication by gates in the optimized circuit which is a basis for delay-insensitivity.

This Section gives a formal ground for justifying these properties.

The effectiveness and correctness of the optimization flow are based on two important facts:

- Two phases of operation (set and reset) are handled separately. In fact optimization is performed only for set phase since the reset procedure is uniform (see the comments to Figure 2)
- During technology-independent multilevel synthesis Design Compiler performs only algebraic operations.

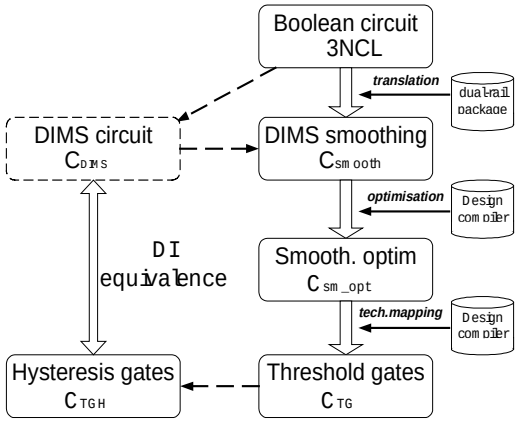

Figure 11. Optimization flow

The overall optimization flow is illustrated in Figure 11. It starts from 3NCL circuit, that apart from 3 value interpretation of Boolean netlist, is of no difference with its Boolean prototype. 3NCL circuit might be translated into dual-rail DI implementation using DIMS technique (the result is denoted by C_{DIMS}). DIMS implementation is further converted into conventional basis by replacing C-elements by AND gates (C_{smooth}). In fact, the transformation from 3NCL into C_{smooth} is performed in one step using a dual-rail package. Though DIMS circuit is an imaginary object in the optimization flow, we like to keep it in Figure 11 for better understanding and establishing the DI equivalence with the final implementation. Design Compiler first optimizes a Boolean network for smoothing circuit C_{smooth} (resulting in C_{sm_opt}) and then maps it into the library

of threshold gates (C_{TG}). Finally Boolean threshold gates of C_{TG} are replaced by threshold gates with hysteresis which gives the implementation C_{TGH}.

The validity of the suggested optimization flow is based on two facts:

1. The known result that DIMS technique is a correct DI implementation for 3NCL [25].
2. The equivalence between C_{TGH} and C_{DIMS} which is established below.

An equivalence between C_{TGH} and C_{DIMS} is proved by separate consideration of set and reset phases. Intuitively: for set phase we prove that each object in optimization flow shows the same behavior, while for reset phase C_{TGH} has a DI behavior which is correct by construction (ensured by inherent properties of gates with hysteresis).

Though the networks for C_{TGH} and C_{DIMS} consist of sequential gates abusing formality we will refer to them as Boolean networks bearing in mind that two phases of network functioning are considered separately.

Our first property states that C_{DIMS} and C_{TGH} have the same input-output behavior.

Property 1. Boolean networks for C_{DIMS} and C_{TGH} are equivalent under two-phase operation.

Proof: The sets of primary inputs and outputs for C_{DIMS} and C_{TGH} are clearly the same. Let us show that for any input pattern the outputs of C_{DIMS} and C_{TGH} coincide.

Case 1. Set phase.

In the beginning of set phase all primary inputs, internal gates and primary outputs of C_{DIMS} are in 0. Inputs under the two-phase operation change monotonously. From unateness of gate functions follows that every gate in a set phase can produce at most one transition from 0 to 1. Under monotonously increasing values on its inputs C-element with the initial value of the output 0 behaves like an AND gate. The latter proves that in a set phase C_{DIMS} and C_{smooth} have the same input-output behavior. Optimization performs only equivalent transformations over the networks and C_{smooth} is equivalent to C_{sm_opt}. In the mapping of C_{sm_opt} into threshold gates with hysteresis the set functions of threshold gates are the same as in C_{sm_opt}. This guarantees that under the initial state 0 for all gates and inputs, and monotonous transitions on inputs, C_{sm_opt} and C_{TG} are equivalent. The latter by transitivity gives an equivalence of C_{DIMS} and C_{TGH} in a set phase.

Case 2. Reset phase.

For reset phase we need to prove that transition of all primary inputs to NULL state (every input is reset to 0) resets every output to 0. This is guaranteed by the properties of threshold gates with hysteresis. Let us levelize the gates in a Boolean network C_{TGH} according to topological ordering [14]: a gate which has only primary inputs at its fan-in goes to Level *1*, a gate with fan-ins from *(i-1)* level or lower goes to Level *i*.

All threshold gates with hysteresis have a set function which is unate and reset function which corresponds to disjunction (actually inversion of disjunction) of all fan-ins (see Figure 2). When primary inputs make transitions to 0 the set function of every gate at Level *1* take 0 value but the gate will keep its output value until the reset function will become 1. This happens when all gate fan-ins are reset in 0 which is acknowledged by resetting gate output. The wave of resets is propagating from level to level and finally reaches primary outputs. Hence C_{DIMS} and C_{TGH} are equivalent in a reset phase as well. ∇

The next statement tells about keeping DI properties of C_{TGH} by claiming that gates with hysteresis of C_{TGH} implementation are strongly indicating.

Property 2. Every gate in C_{TGH} network implementation is strongly indicating.

Proof: Let us consider set and reset phases separately.

Case 1. Set phase.

1) C_{smooth} is **prime and irredundant**. Indeed, under the topological ordering of gates, C_{smooth} (obtained from C_{DIMS}) has the following structure: AND gates at odd levels and OR gates at even, where each AND gate is unate and none of them cover each other. From this follows that AND gates are prime and irredundant. AND gates in the fan-in of a particular OR gates represents minterms and hence during the operation of C_{smooth} fan-ins of OR gates are always orthogonal. The latter means the primality and irredundance of C_{smooth} with respect to OR gates as well.

2) Prime and irredundant networks are known to be 100% stuck-at testable [38].

3) In [39] it was proved that under the algebraic transformations the set of multifault test vectors is maintained. Moreover if the circuit obtained by algebraic transformations is irredundant then the set of test vectors for single stuck-at faults is also maintained. Hence we can conclude that C_{smooth} and C_{sm_opt} have the same sets of test vestors for single stuck-at faults.

4) Let us consider gates of C_{sm_opt} in topological order. Suppose that gate *g* at the first level of the network is not strongly indicating with respect to some input *a*. Algebraic transformations preserves unateness and hence

we can represent a logic function for g as: $F_g = a H + C$, where C and H are not dependent from variable a. a is not indicated if and only if there exists a set X of primary inputs such that $a=1$ in X and $H(X)=1$, $C(X)=1$. Clearly that in C_{smooth} vector X is a test for single stuck-at by input a. Therefore we can include X in the set of test vectors for C_{smooth}. However X is not testing single stuck-at a for a C_{sm_opt} and we obtain the contradiction with (3). Considerations similar to [39] can be repeated to internal (upper levels) gates of C_{sm_opt} as well. Hence C_{sm_opt} is strongly indicating in a set phase.

*5)*It was shown in [39] that tree-based mapping also maintains the set test vectors for stuck-at faults. Then repeating arguments from (4) we might conclude that C_{TG} (and C_{TGH} respectively) is strongly indicating in set phase.

Case 2. Reset phase. The strong indicatibility of C_{TGH} in reset phase directly follows from the properties of threshold gates with hysteresis (see the proof of Property 1). ∇

7. Experimental results

The results obtained through the suggested approach are compared in two ways.

First, we compare the synthesis netlist to NCL designs entered with schematic capture. Since our goal is to increase our design productivity by using synthesis, this is an important measure. The numbers on manual designs are taken from the Theseus Logic design division, which is experienced in NCL implementations. They have designed and fabricated 18 chips (five of them over 150,000 transistors with the biggest one of 660,000 transistors). The synthesized designs are produced by using Design Compiler from Synopsys. **Error! Reference source not found.** below shows the comparison. Area results are in number of transistors for a semi-static prototype library.

Table 1. NCL manual vs. synthesized designs.

Circuit	Manual	Synthesis	ratio
and4	66	68	103%
test7	140	126	90%
clipper	339	212	63%
and16	352	336	95%
set_cnt	238	198	83%
case	594	482	81%
bit_cnt	1059	1072	101%
sync_state	1008	814	81%
sum	4346	3380	78%

With the exception of two circuits, the implementations obtained by Design Compiler are excellent. Design Compiler does a good job optimizing NCL circuits and should be extremely useful in our design flow.

The second area of comparison is NCL against synchronous designs. The NCL MUX (from Section 5) requires 44 transistors against 14 transistors of synchronous implementation, which is 3.1 times larger. This penalty is similar to a Viterbi decoder that we translated straight from the clocked design (See **Error! Reference source not found.**). The results are better if we adapt the code to NCL. For example, we downloaded benchmarks used to compare HDL synchronous synthesis tools from different vendors [40]. The first one, hostfird is a thirty-two by 16-bit unidirectional FIFO buffer. By redesigning it, we significantly reduced the area because asynchronous designs are ideally suited to FIFOs, whereas the clocked design requires two counters, empty and full logic, two decoders, and two multiplexers. The second design, addrconv compares favorably. The clocked data used the same version of Design Compiler and a library based on LSI Logic's LCB500K library [41]. The gate count is the number of actual gates, not the equivalent gate count, a number often used for comparison and is typically four transistors per gate.

Table 2. Synchronous vs. NCL designs.

	gates			transistors		
module	clock	NCL	ratio	clocked	NCL	ratio
decoder	1010	2344	2.3	8788	30574	3.5
hostfird	1691	1123	0.7	23498	26500	1.1
addrcon	41	70	1.7	524	852	1.6

A more than three times penalty in transistors when comparing NCL and synchronous implementations is not surprising. A dual-rail implementation immediately leads to circuits that are twice the size; the rest of the area increase is due to effective delay-insensitivity that needs to be ensured in NCL circuit (though by relaxing DI requirements (using timing assumptions e.g.) one can make area trade-offs).

In the suggested design flow the area penalty can be large. This could be unacceptable for a stand-alone design. However, the prediction [42] tells that in the future ages of SoC most of the chip area (up to 70-75%) will be occupied by memories. For that case a 3 times increase in the rest of circuitry could be an acceptable trade-off for ease and speed of design flow.

We can reduce the transistor count by designing circuits that fit NCL's features better. For example, in the **if-then-else** case, we can reduce the transistor count in one of two ways. If the input data is orthogonal, then we can use an OR gate instead of a multiplexer. This solution requires less logic than the regular Boolean case (MUX). If we compromise the delay insensitivity by allowing isochronic forks on the latch inputs we can use a multiplexer that requires 24 transistors instead of 44, making it 1.7 times larger than the synchronous counterpart, not 3.1.

8. Conclusions and future work.

Theseus Logic is just beginning the development and application our NCL synthesis tools. The conclusion at this stage of our work is simple: we have a working design flow for NCL based on commercial synchronous HDL tools. We can design asynchronous (NCL) circuits as easily as one can design traditional synchronous designs. The results of the synthesis are acceptable, at least by our customers [43].

The main concerns for us now are verification of the timing assumptions behind the isochronic forks ("orphans" in NCL terminology) and reducing the area overhead.

At our current level of technology, (CMOS 0.25μm) all orphans in our designs proved to be safe by simulation. For future technologies, Theseus Logic is developing a special CAD tool to discover dangerous orphans.

There are several ways to reduce area without losing delay insensitivity. For example, we typically get a large area penalty if we generate asynchronous (NCL) circuitry from synchronous RTL code that has been modified only so it can be synthesized. However, the area can frequently be reduced through redsign. For example, an asynchronous design could be less global that synchronous (each register has a separate request instead of a clock). As mentioned in the if-then-else example, a more detailed consideration of register interactions can notably reduce area. One of our future tasks is a HDL preprocessor that could automatically replace some of the typical RTL constructions (like the if-then statement that infers a MUX) by a more effective NCL implementation.

There are several more technical issues that we will address in the synthesis flow in the near future.

- Write DesignWare components to improve performance for arithmetic units by instantiating hand-designed components.
- Adapt the flow to handle Verilog. Currently, we can synthesize Verilog code, but not simulate it.
- We do not want to write a tool to optimize the netlist, since Design Compiler does its job well. However, we would like to reduce the netlist using simple peephole optimizations. We have already started the process and are able to merge gates used for registration with their input gates.

Acknowledgment. We thank Luciano Lavagno for useful comments on the draft of this paper.

REFERENCES

1. Scott Hauck. Asynchronous design methodologies: An overview. Proceedings of the IEEE, 83(1):69-93, January 1995.

2. C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. Proceedings of the IEEE, 87(2):223-233, February 1999.

3. Beth Martin. Technology 1999. Analysis and forecast issues. Electronic Design Automation. IEEE Spectrum January 1999 Volume 36 Number 1.

4. Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In Proc. European Conference on Design Automation (EDAC), pages 384-389, 1991.

5. J. Cortadella, M. Kishinevsky, A.Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions on Information and Systems, E80-D(3):315-325, March 1997.

6. K.Yun, P. Beerel, V. Vakilotojar, A. Dooply, J. Arcco. The Design and Verification of A High-Performance Low-Control-Overhead Asynchronous Differential Solver, In Proc. of International Symposium on Advanced research in Asynchronous Circuits and Systems, The Netherlands, pp 140-153, 1997

7 . Kees van Berkel, Hans van Gageldonk, Joep Kessels, Cees Niessen, Ad Peeters, Marly Roncken, and Rik van de Wiel. Asynchronous does not imply low power, but .... In Anantha Chandrakasan and Robert Brodersen, editors, Low Power CMOS Design, pages 227-232. IEEE Press, 1998.

8. W.A. Lien, P.Day, C.Faransworth, D.L.Jackson, J.Liu, and N.Paver. Noise in a self-timed and synchronous implementation of a DSP. Unpublished, White Paper, Cogency Technology Inc., 1997.

9. C. Shannon. A Symbolic analysis of relay and switching circuits. Trans. AIEE, Vol.57 (1938), pp. 713-723.

10. Stephen H. Unger. The Essence of Logic Circuits. IEEE Press, second edition, 1997.

11. D. Sylvester, K. Keutzer. Getting to the bottom of Deep Submicron, In Proc. Of International Conference on Computer-Aided Design, ICCAD-98, pp 203-211, 1998
12. Steven M. Nowick and David L. Dill. Automatic synthesis of locally clocked asynchronous state machine. In Proc. International Conf. Computer-Aided Design (ICCAD). IEEE Computer Society Press, pp.318-321, November 1991.
13. A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, and L. Lavagno. The use of Petri nets for the design and verification of asynchronous circuits and systems. Journal of Circuits Systems and Computers, 8(1):67-118, 1998.
14. R. K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. Proceedings of the IEEE, Vol.78, No.2, February 1990, pp,264-300
15 . Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, Developments in Concurrency and Communication, UT Year of Programming Series pages 1-64. Addison-Wesley, 1990.
16 . T. Verhoeff. Delay-insensitive codes – An overview. Distributed Computing, vol. 3, no 1, pp. 1-8, 1988.
17. Karl M. Fant and Scott A. Brandt. NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis. In International Conference on Application-specific Systems, Architectures, and Processors, pages 261-273, 1996.
18. Gerald E. Sobelman and Karl Fant. CMOS circuit design of threshold gates with hysteresis. In Proc. International Symposium on Circuits and Systems, pages 61-64, June 1998.
19. Victor I. Varshavsky, editor. Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.
20. C. Sims and H. J. Gray. Design criteria for autosynchronous circuits. In Proc. Eastern Joint Computer Conf. (AFIPS), volume 14, pages 94-99, December 1958.
21. Zvi Kohavi. Switching and Finite Automata Theory. McGraw-Hill, 1978.
22. M. L. Dertouzos. Threshold Logic: A Synthesis Approach. MIT Press, 1965.
23. Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, Introduction to VLSI Systems, chapter 7. Addison-Wesley, 1980.
24. Ilana David, Ran Ginosar, and Michael Yoeli. An efficient implementation of Boolean functions as self-timed circuits. IEEE Transactions on Computers, 41(1):2-11, January 1992.
25. Jens Sparso and Jorgen Staunstrup. Delay-insensitive multi-ring structures. Integration, the VLSI journal, 15(3):313-340, October 1993.
26. Christian D. Nielsen. Evaluation of function blocks for asynchronous design. In Proc. European Design Automation Conference (EURO-DAC), pages 454-459. IEEE Computer Society Press, September 1994.
27. Daniel H. Linder and James C. Harden. Phased logic: Supporting the synchronous design paradigm with delay-insensitive circuitry. IEEE Transactions on Computers, 45(9):1031-1044, September 1996.
28. Mark Dean, Ted Williams, and David Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In Carlo H. Sequin, editor, Advanced Research in VLSI, pages 55-70. MIT Press, 1991.
29. Peter Vanbekbergen, Albert Wand, and Kurt Keutzer. A design and validation system for asynchronous circuits. In Proc. ACM/IEEE Design Automation Conference, June 1995.
30. M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 135-144, April 1999.
31. N.Starodoubtsev, A. Yakovlev and S. Petrov. Use of VHDL environment for interactive synthesis of asynchronous circuits Proceedings VHDL Forum in Europe Spring'96 Working Conference, Dresden, Germany, May 1996, pp. 21-33.
32. Ivan Blunno and Luciano Lavagno. Towards a language-based design flow for asynchronous circuits. In Proc. International Workshop on Logic Synthesis, June 1999.
33. Model***Sim*** from Model Technology - http://www.model.com/
34. A.Rushton. VHDL for Logic Synthesis. 2-nd ed., J.Wiley, 1998.
35. Steve Carlson. Introduction to HDL-based Design using VHDL. Synopsys Inc, 1991. See also - Design Compiler 99 - http://www.synopsys.com/products/logic/logic.html
36. Exemplar's LeonardoSpectrum - http://www.mentorg.com/leonardo_spectrum/index.html
37. Envisia Ambit synthesis tool - http://www.cadence.com/technology/chip/products/
38. D.Scherz, G. Metze. On the design of multiple fault diagnosable networks. IEEE Trans. Comput., vol. C-21, August, 1972
39. G. Hatchel, R. Jacoby, K. Keutzer, C. Morrison. On properties of Algebraic Transformations and the Synthesis of Multifault-Irredundant Circuits. IEEE Transactions on CAD, vol.11, N 3, pp. 313-321, 1992
40. Brian Dipert. Hands-On Project: Synthesis Shoot-Out at the EDN Corral**.** EDN magazine
Issue 19: Sept 11, 1998 http://www.ednmag.com/ednmag/reg/1998/091198/19df2.htm
41. LCB500K Preliminary Design Manual, LSI Logic Corporation, Milpitas, CA, 1995.
42. International Technology Roadmap for Semiconductors, 1999 edition. - http://www.itrs.net/1999_SIA_Roadmap/Home.htm
43. David Lammers. Motorola taps data-driven logic from Theseus for SoC design. EE Times, 10/22/99, p.p.1,6.