

NCL_X design flow: Fibonacci Series Machine as a Case Study

Itamar Cohen,
Department of Electrical Engineering,
Technion, Israel Institute of Technology.
ofanan@tx.technion.ac.il
Spring 2005

Abstract

In spite of the many advantages of asynchronous methodologies in terms of power, timing, EMI and noise immunity, they are still not widely accepted in the industry, mainly due to poor CAD support.

NCL_X is a design flow, which was suggested in aim to close this gap, and enable the design of asynchronous digital circuits by standard commercial CAD tools.

In this paper I review the papers which originally suggested NCL_X design flow, and use an asynchronous Fibonacci series calculator as a case study for evaluating the benefits of NCL_X in terms of area, power, timing and complexity of design. Finally, I conclude and refer to NCL_X within the wider context of design methodologies of asynchronous circuits.

List of acronyms

CAD – Computer Aided Design

DICLASP – Delay Insensitive Carry Look-ahead Adder with Speedup

DRN – Double-Rail Network

DR-ST – Double-Rail Self-Timed

EMI – Electro-Magnetic Interference.

FA – Full Adder

HA – Half Adder

HW - Hardware

LCC – Longest Carry Chain

NCL – Null Convention Logic

NCL_X – Null Convention Logic with eXplicit completion

(Q)DI – (Quasi)-Delay Insensitive

RCA – Ripple Carry Adder

THmn – ThresHold m-of-n inputs gate

TP – throughput

Table of contents

1.0 Theoretical background	3
1.1. Summary of Null Convention Logic (NCL).....	3
1.2. NCL with explicit completion (NCL_X).....	4
1.3 Design of an asynchronous Fibonacci series calculator	7
2.0 Architecture and Implementation issues.....	8
2.1 C-element.....	9
2.2 Completion detector.....	9
2.3 NCL register	9
2.4 NCL_X pipe level.....	11
2.5 The adder: high-level design	11
2.5.1 Possible optimizations for the adder's architecture	12
2.6.0 The adder: low-level design and the completion network.....	12
2.6.1 FA: Straight-forward implementation	12
2.6.2 Discussion: can the completion network be optimized?.....	13
2.6.3 Discussion: does NCL_X fulfill the QDI demands?	14
2.6.4 Optimized stable adder	14
2.7.0 The top design.....	15
2.7.1 The number of pipe levels	15
2.8.0 Functional verification of the design	16
2.9.0 Synthesis code Vs. Simulation code in NCL_X.....	16
3. Qualitative Comparison: NCL_X Vs. other asynchronous design flows	17
4. Conclusions.....	18
References.....	18
Appendix A: Clarification mails from the papers' authors.	19
Appendix B: Transistor-level implementation of C-element in Verilog.	20
Appendix C: ReadMe for the code files	21

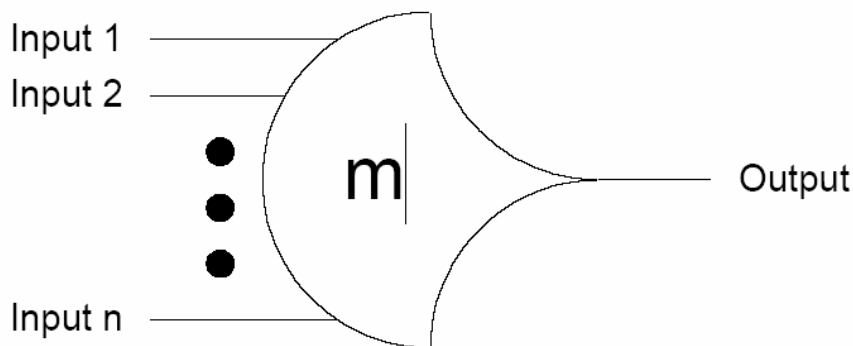
1.0 Theoretical background

The introduction below assumes acquaintance of the reader with the principles of asynchronous design. Thus, only brief summary of the relevant issues is given, and references for further given are given inside the relevant subchapters.

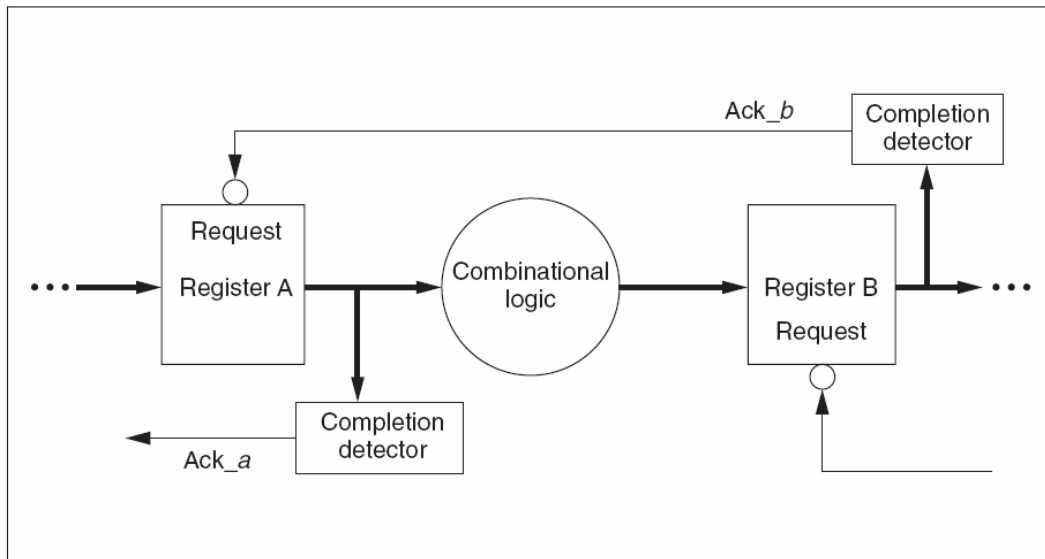
1.1. Summary of Null Convention Logic (NCL)

The basic element of *NCL* [1] logic design is the *m-of-n threshold gate* (*TH_{mn} gate*) with hysteresis, i.e. a gate, whose output is set high when at least *m* out of its *n* inputs are high, and set low when **all** its inputs are low. Note, that a 1-of-*n* threshold gate is an OR gate, and *n*-of-*n* threshold gate is a *C-element*.

Below is a diagram of a TH_{mn} gate:



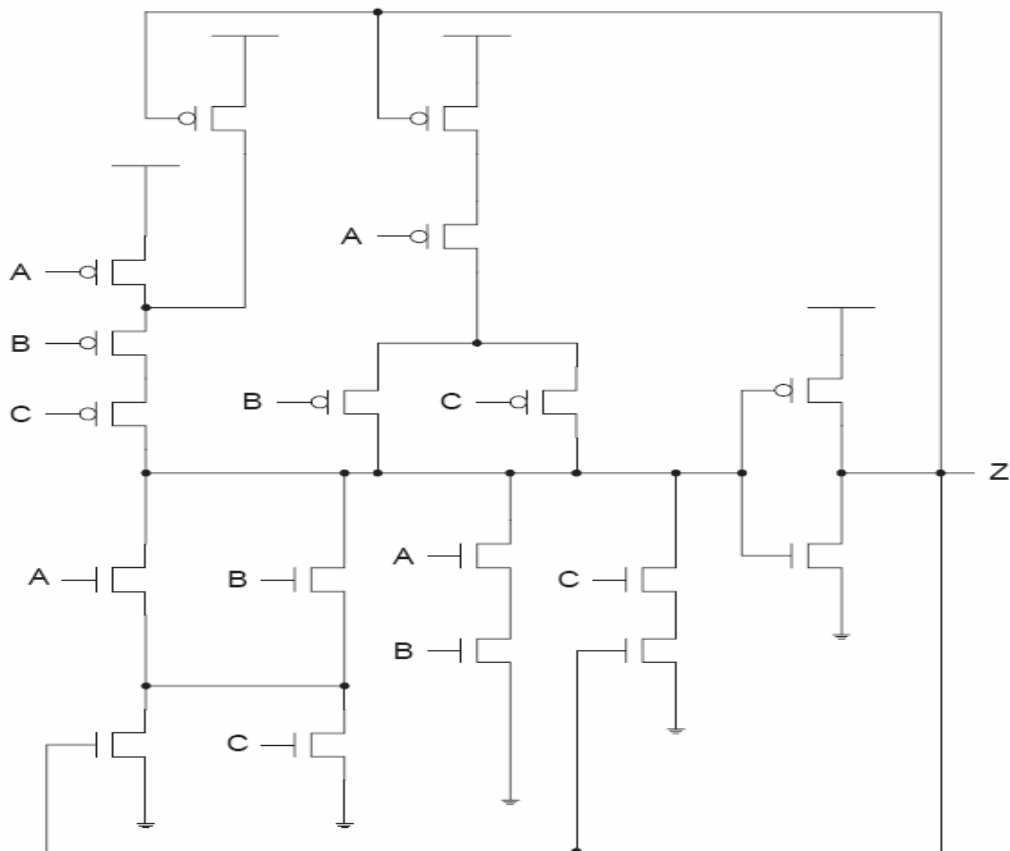
NCL deploys Mutually Exclusive Assertion Groups (MEAGs) such as *dual-rail*, *quad-rail* or others, to incorporate data and control information into one mixed path. *NCL* follows so-called “*weak condition*” of Seitz’s delay-insensitive signaling scheme [11]. *NCL* is *quasi-delay-insensitive*, meaning that forks in wires within basic components are assumed to be *isochronic*. Note, that as the class of (full) Delay-Insensitive (DI) circuits is limited and impractical, most circuits that are referred to in the literature as DI are actually QDI [BOOK, subchapter 2.5.2]. *NCL* employs a *completion component* at the output of each *NCL register* [4]. The scheme below demonstrates *NCL*’s pipelining using these completion components:



A very thorough and clear description of NCL is found in [4].

1.2. NCL with explicit completion (NCL_X)

NCL_X [2] aims to solve a major drawback of the usual NCL design flow – the highly irregularity of THmn gates. For instance, below is a figure of a static implementation of TH23 gate:



The usage of a general THmn gate of arbitrary m, n values is not trivial, as such gates aren't included in the technology libraries of standard CAD tools, and standard HDL

tools don't well support transistor-level writing (VHDL does not enable any transistor-level design; Verilog enables transistor-level coding, but the transistors are only partially supported by simulation tools, and are not support by synthesis tools; see appendix B).

In addition, NCL methodology doesn't supply a general straight-forward flow for translating SPEC into HDL code. Indeed, such a translation is not trivial even when dealing synchronous design; yet, the many-years experience and the good support of the simulation and verification tools enable the design of very complex synchronous circuits, while even a well-skilled VLSI engineer would need a long learning phase until being able to design a rather simple NCL circuit.

NCL_X attacks this problem by decreasing the usage of THmn gates for $1 < m < n$. In fact, for many designs the only non-standard gates which are used by NCL_X are C-elements. This methodology enables design, simulation and synthesis by standard CAD tools, with the exception of the C-elements, which are almost a must when dealing asynchronous design.

Note: in [2], [3] no explicit referring to the type of library cells needed for the NCL_X flow is given. The paragraph above is my based on the clarification mails said by one of the authors (see appendix A).

The simplicity of the design comes at the cost of additional circuitry for ensuring the delay-insensitivity, which is not guaranteed anymore when moving from TH gates to standard gates. In practice, each pair of dual gates needs a *completion detector*, and the *completion network* of each pipelining level assert the *done* signal high (low) only after **all** the inputs, outputs and internal signals in it have reached the DATA (NULL) state.

NCL_X design flow consists of 4 steps:

- A. Synthesize (and optimize) the RTL SPEC to standard GTech library, which implements the set values of the outputs.
- B. Reduce the logic network to unate gates, by using two different variables, $a.1$ and $a.0$ for the direct and inverse values of signal a .
- C. Expand the combinational logic by creating a dual network, which would produce the invert values of the output.
- D. Ensure the delay insensitivity by providing local completion detectors (OR gates) for each pair of dual gates and for each pair of dual and inverse values of the input signals. Then, connect the outputs of all the completion detectors (notated by "go") into a completion network (multi-input C-element) with a single output *Done*.

Example:

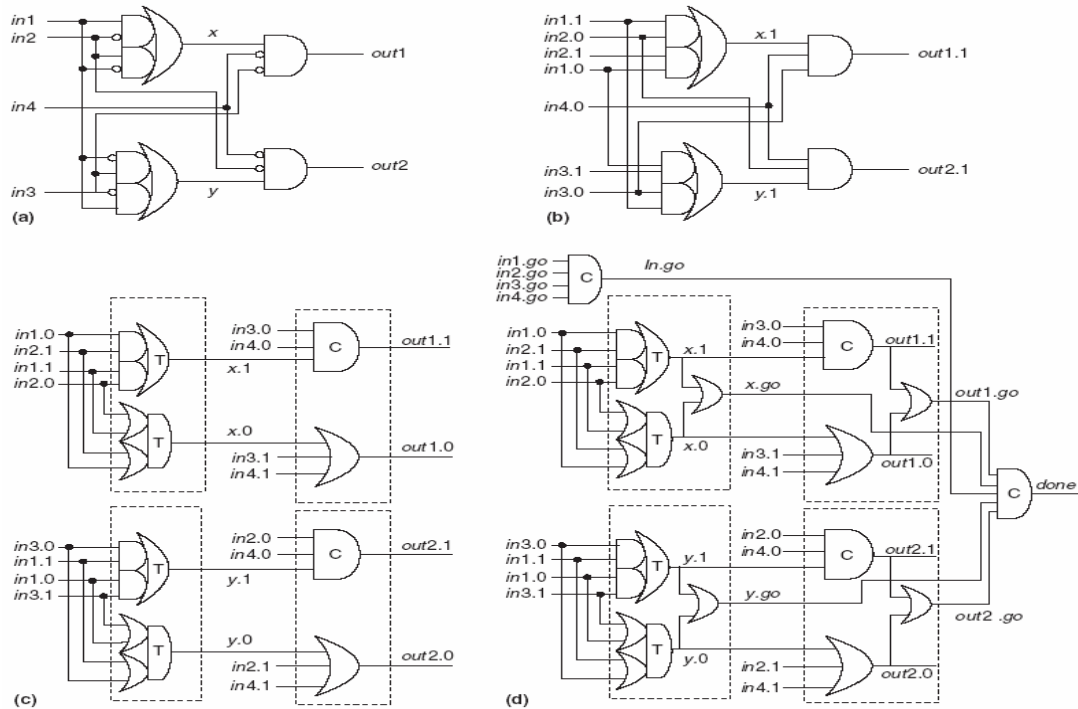
Below is the RTL specification for a 4-to-2 encoder; below it, the figures a, b, c, d demonstrate the steps described above.

```
encode : process(din)
begin
  if din = "1000" then
    d <= "11";
  elsif din = "0100" then
    d <= "10";
  elsif din = "0010" then
    d <= "01";
  elsif din = "0001" then
```

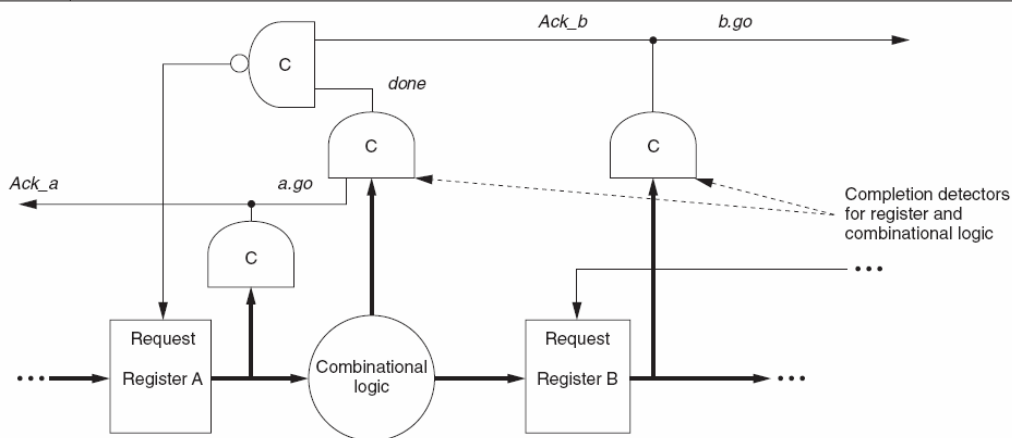
```

d <= "00";
else
d <= (others => '0');
end if;
end process encode;

```



The diagram below shows the system-level NCL_X design.



NCL_X suggests separation between the simulation model and the synthesis model. For simulations, one should instantiate NCL registers and model the sequential behavior of NCL gates; for synthesis, threshold gates are represented by their set functions, and look like Boolean gates.

These suggestions raise some questions, which are detailed and discussed in chapter 2.

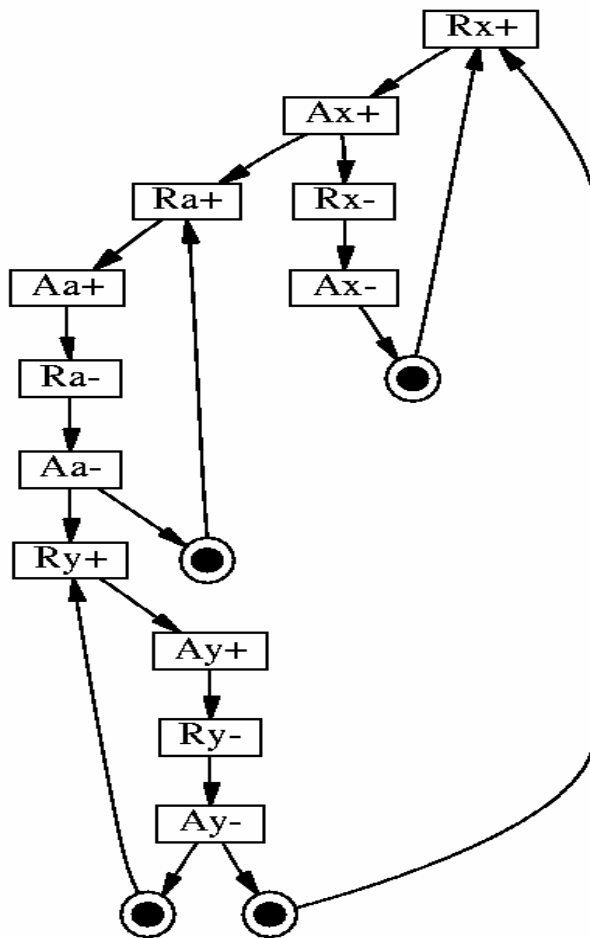
A proof that NCL_X architecture is DI is given at [3].

$F(k-2)$, will be noted by Y . The number of registers in X will be noted as `NUM_OF_PIPE_LVL`S. For a correct and simple execution of the circuit, we will assume that the number of X registers is equal to the number of Y registers.

Note: The “Starter” is given only for better understanding and for simulations; it is NOT part of the design, though – it wasn’t implemented by `NCL_X` method, neither was it synthesized.

At a higher abstraction level, either X , Y and the adder, A , may be represented by a single asynchronous register, with “*Req*” and “*Ack*” lines; for the adder, the *Req* and *Ack* lines are in practice the “*go*” and “*done*” lines respectively.

In that case, the Petri Net STG (State Transition Graph) of the circuit is:



Functional verification of the protocol by a behavioral level Verilog code is found at the attached [HW3].

2.0 Architecture and Implementation issues

In the VHDL implementation, standard Boolean gates (with no hysteresis) are modeled (for simulation) as 0-delay units.

All other gates – C-elements, or larger entities, which contain C-elements within them – are asynchronous sequential components, and thus are assigned a user-defined delay.

Other parameters, such as the number of pipe levels (see discussion later in this chapter) and the number of data bits, can also be manipulated.

Below is an overview of the various units in the design. Wherever relevant, optional optimizations and the implications of using the NCL_X in comparing to other design flows are discussed.

The full VHDL source code is attached, and appendix C contains a ReadMe file regarding it.

Unless stated differently, all the synthesis figures present the synthesized entity **after default optimization** of **medium** effort. This stands in a line with the detailed description of NCL_X in [2], and especially with the need to use complex gates (see 2.6).

2.1 C-element

As usual NCL_X (see 1.2), C-elements were implemented in two different architectures: a behavioral architecture - for simulation (if synthesized, this architecture has an inferred latch inside); and an architecture for synthesis, which is actually built upon a series of AND gates, which represent the *set* function of the C-element.

Note, that such a description of a C-element doesn't well imitate neither its functional behavior (as a cell, which has memory); nor its transistor-level implementation. However, an n-inputs AND gates has roughly the same transistor count as an n-inputs C-element. "Roughly", because A) C-element contains one more inverter between its output and its negate for stability; and B) cascading a few C-elements, which have serial paths at both the p-network and the n-network, may be better done in a different way than cascading a few AND gates.

The C-element is of *generic* number of bits and delay.

Possible optimizations

As shown in [4], an n-input C-element may be built by hierarchical cascading of $\log_4(n)$ levels of 4-inputs C-elements. Such a construction may minimize the delay.

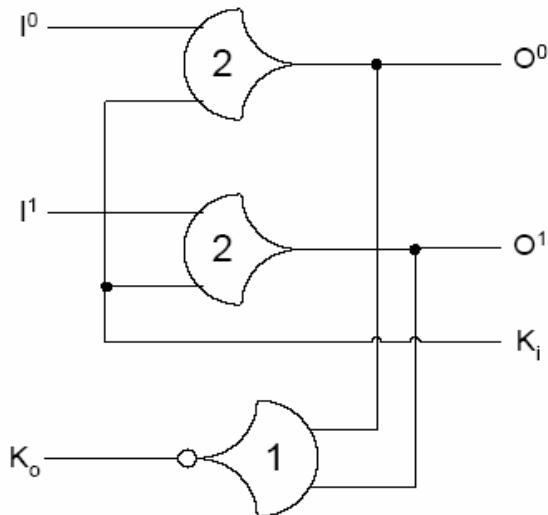
However, as the Fibonacci machine is built upon various input-width C-elements, including widths which are not a power (neither even a multiplication) of 4, this optimization was not used. Note also, that this optimization is orthogonal for the usage of this or other design scheme, i.e. it doesn't help evaluating NCL_X.

2.2 Completion detector

The completion detector is built by ORing each pair of dual-rail input, and driving the ORs outputs into a multi-input C-elements.

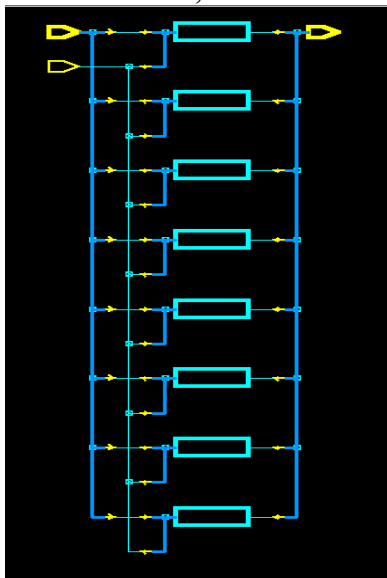
2.3 NCL register

NCL register was implemented as described in [4] – see the figure below.



In the terminology of [4], K_i , K_o are parallel to *req*, *ack* respectively in our notation. As in NCL_X the completion detection and the control mechanism slightly differ from those in other NCL design scheme, *NCL_reg* was implemented merely as the register itself (which, informally speaking, functions like an asynchronous version of a latch), while the unit which contains the register **plus** its control is *NCL_X_pipe_lvl*. *NCL_reg* is actually no more than a pair of 2-inputs C-elements per each dual-rail data-input; each C-elements receive *Req* and one data bit as input, and outputs the corresponding output data bit.

In the synthesis figure below the boxes represent C-element. The input & output buses are of data bits, and the common single-bit input is *Req*.

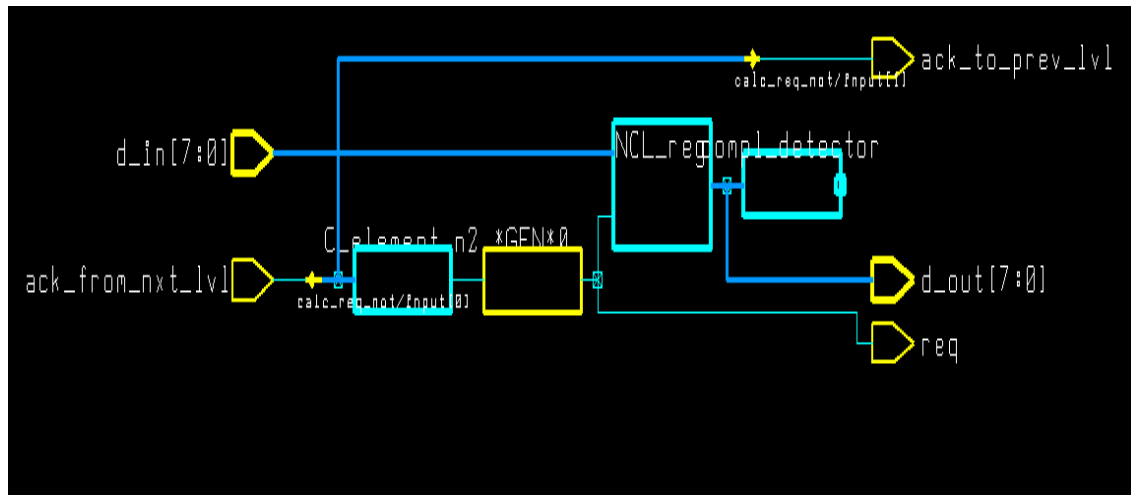


Note, that C-element, the completion detector and the NCL register used in NCL_X are identical to those used by other NCL implementations.

2.4 NCL_X pipe level

This entity represents a pipe level where there is no combinational logic between the current pipeline level and the next one. It's built of an NCL register, plus C-elements which calculate the *req* and *go* signals. The *ack* output to the previous level is actually the *go* signals, obtained by a C-elements, whose inputs are the register's outputs. As no combinational logic, *done* is simply shorted to *go*.

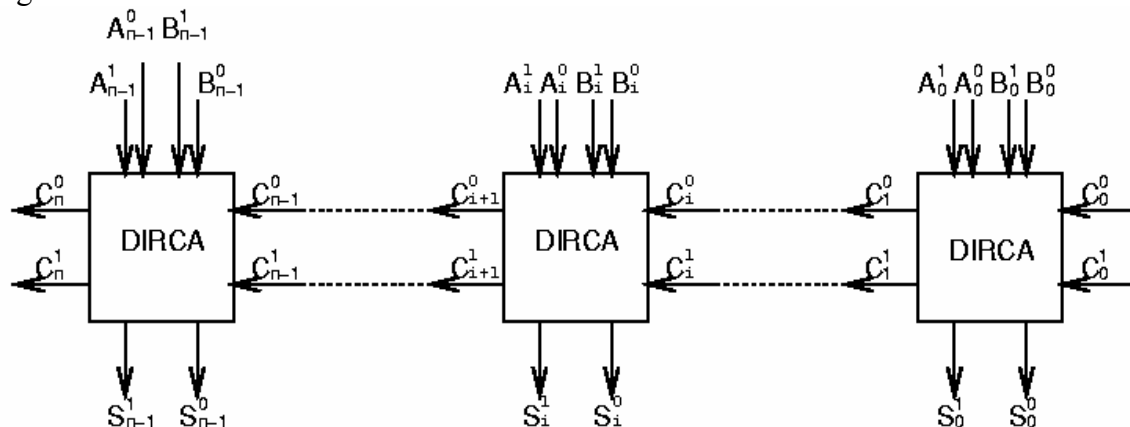
In the picture of the synthesized NCL-X pipe level below, the yellow box is a negligible connector. The large pale blue box is the NCL register, and the smaller ones are the C-elements which calculate the *go* and *req* signals.



In fact, an NCL_X pipe level is almost identical to a pipe level with no combinational logic also in general NCL design (up to a different nomenclature).

2.5 The adder: high-level design

The adder which was built is DIRCA (DI Ripple Carry Adder), which is shown in the figure below:

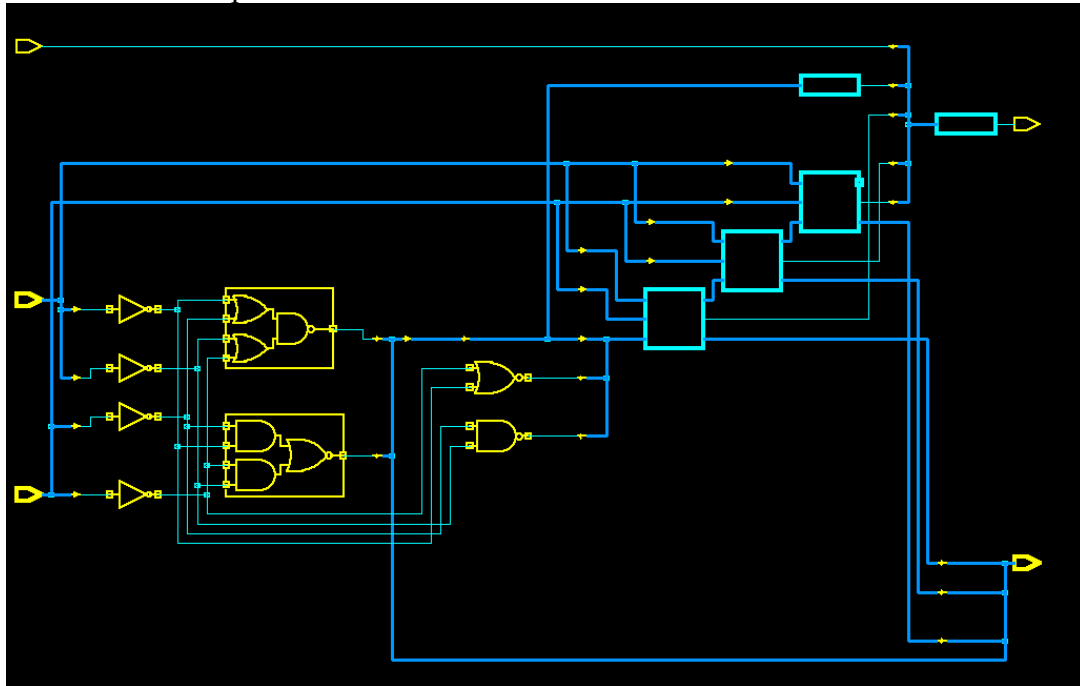


The adder's completion detector receives as input all the sum bits and all the FAs' *done* signals, as discussed in details below.

The adder's delay is mainly a function of the longest carry chain (LCC). Simulations and combinatory analysis show that for a uniformly distributed input, the expected LCC is approximately $\log(n)$. However, usually the input is not uniformly distributed.

Calculations of the probability distribution function of this adder's delay are found in [6].

Below is a figure of the adder. Note, that for saving area (on behalf of the regularity of the design), the first FA was replaced by a HA, as for our purposes we need adder with no carry-in bit. The square boxes are the FAs, and the other pale blue boxes are C-elements / completion detectors.



2.5.1 Possible optimizations for the adder's architecture

While DIRCA is merely an asynchronous version of the simple RPA, more sophisticated adders' architectures enable speeding the calculation without increasing the asymptotic area consumption. An example of such an adder is DICLASP (DI Carry Look-ahead Adder with Speedup) [7], which obtains average delay of $O(\log(\log(n)))$, at an area consumption of only $O(n)$ (truly, the coefficients for the area consumption are probably larger than those of DIRCA).

Indeed, NCL_X doesn't obligate the usage of a specific adder. Discussion about different asynchronous adders' architecture is beyond the scope of this paper.

2.6.0 The adder: low-level design and the completion network

2.6.1 FA: Straight-forward implementation

The FA was written using 4 different architectures, which represent the 4 steps in NCL_X design flow, as described in 1.2.

First step: represent a GTech mapping for the FA. As noted in 1.2, the (structural) VHDL architecture should better include no complex gates which are not easily reducible to unate gates; e.g. it shouldn't include a XOR gate, though such a gate may optimize the delay of the block in the first step. Here's the VHDL code:

```
not_a    <= not (a(1));
not_b    <= not (b(1));
not_c_in <= not (c_in(1));
```

$$\begin{aligned}
c_out_sig(1) &\leq (a(1) \text{ and } b(1)) \text{ or } (a(1) \text{ and } c_in(1)) \text{ or } (b(1) \text{ and } c_in(1)); \\
s_sig(1) &\leq (a(1) \text{ and } b(1) \text{ and } c_in(1)) \text{ or } (a(1) \text{ and } \text{not } b \text{ and } \text{not } c_in) \text{ or} \\
&\quad (\text{not } a \text{ and } b(1) \text{ and } \text{not } c_in) \text{ or } (\text{not } a \text{ and } \text{not } b \text{ and } c_in(1));
\end{aligned}$$

Second step: reduce the design to unate gates by using $a.0$ for the inverse value of signal a :

$$\begin{aligned}
c_out_sig(1) &\leq (a(1) \text{ and } b(1)) \text{ or } (a(1) \text{ and } c_in(1)) \text{ or } (b(1) \text{ and } c_in(1)); \\
s_sig(1) &\leq (a(1) \text{ and } b(1) \text{ and } c_in(1)) \text{ or } (a(1) \text{ and } b(0) \text{ and } c_in(0)) \text{ or} \\
&\quad (a(0) \text{ and } b(1) \text{ and } c_in(0)) \text{ or } (a(0) \text{ and } b(0) \text{ and } c_in(1));
\end{aligned}$$

Third step: – expand the logic by creating a dual network, which would produce the invert values of the output.

$$\begin{aligned}
c_out_sig(1) &\leq (a(1) \text{ and } b(1)) \text{ or } (a(1) \text{ and } c_in(1)) \text{ or } (b(1) \text{ and } c_in(1)); \\
s_sig(1) &\leq (a(1) \text{ and } b(1) \text{ and } c_in(1)) \text{ or } (a(1) \text{ and } b(0) \text{ and } c_in(0)) \text{ or} \\
&\quad (a(0) \text{ and } b(1) \text{ and } c_in(0)) \text{ or } (a(0) \text{ and } b(0) \text{ and } c_in(1));
\end{aligned}$$

$$\begin{aligned}
c_out_sig(0) &\leq (a(0) \text{ or } b(0)) \text{ and } (a(0) \text{ or } c_in(0)) \text{ and } (b(0) \text{ or } c_in(0)); \\
s_sig(0) &\leq (a(0) \text{ or } b(0) \text{ or } c_in(0)) \text{ and } (a(0) \text{ or } b(1) \text{ or } c_in(1)) \text{ and} \\
&\quad (a(1) \text{ or } b(0) \text{ or } c_in(1)) \text{ and } (a(1) \text{ or } b(1) \text{ or } c_in(0));
\end{aligned}$$

Fourth step: add completion detectors for ensuring the DI of the design. This issue is widely discussed herein.

2.6.2 Discussion: can the completion network be optimized?

As described in [2], a close examination of the Boolean functions in the circuits may enable significant area saving, by realizing that some inputs of the completion network are actually acknowledged by other signals, and thus may be omitted without hurting the DI behavior of the circuit.

It would seem very promising, thus, to omit from the completion network all the **internal** carry signals, as the expanded architecture in the 3rd step verifies that no *sum* bit would be DATA before its *carry-in* bit is DATA.

Unfortunately, this intuition is misleading, as the parallel claim doesn't hold for the DATA → NULL switching. In other words, there exists a scenario, in which the both outputs of a FA (and thus, of the whole adder) may switch to NULL while its carry-in bit is still DATA.

Here's an example of such a scenario:

Suppose that both A_n and B_n switch from DATA to NULL.

By using the equations of the 3rd step it's easy to observe that S_n and C_{n+1} would also switch to NULL, regardless of the value of C_n . This is to say that both the sum vector and the carry-out bit may reach NULL while C_n is still DATA, and, as a consequence, *done* signal may be set low, implying 'ready for data'[4]), while C_n is still DATA.

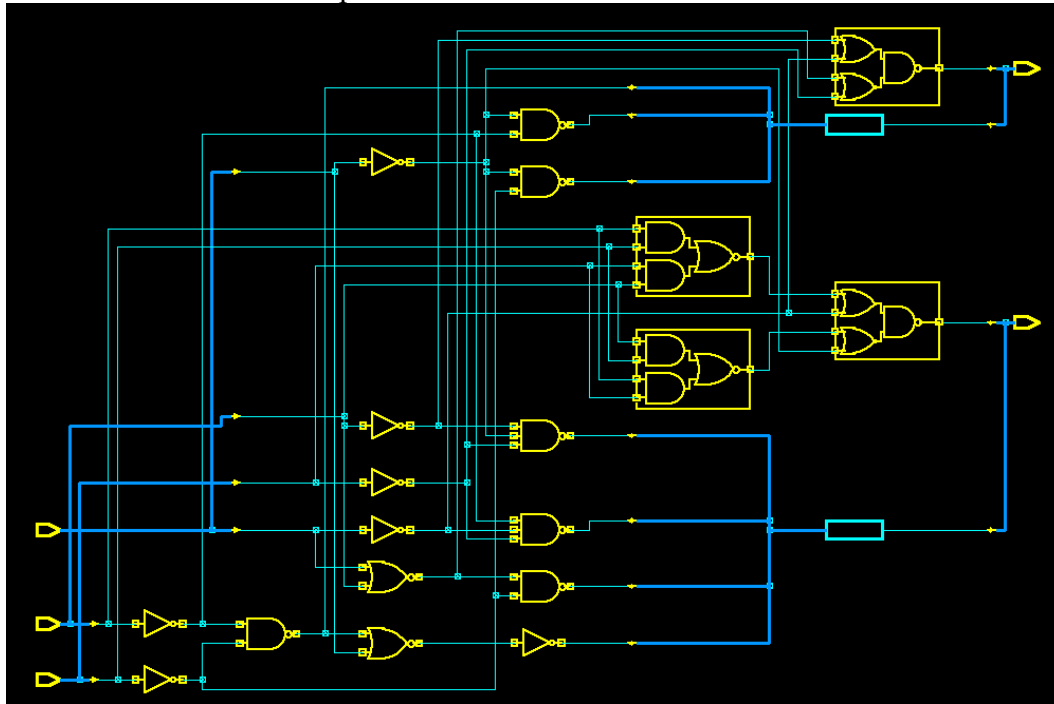
Now, if the gates which calculate C_n are very slow, the next input vectors A , B , and, in particular, A_n and B_n may become DATA again while C_n still holds the DATA of **the previous cycle**, thus causing a disaster.

For preventing such a scenario, **all** the internal carry signals must also be connected to the completion network.

2.6.3 Discussion: does NCL_X fulfill the QDI demands?

The (quasi) DI assumption says that wire forks within the same primitive blocks are *isochronic*. However, no timing assumptions may be taken regarding wires **outside** primitive blocks. This is to say that we may use the simple implementation of a FA which was presented above, only if each primitive gate's output is acknowledged at the FA's output.

As can be seen in the figure of the **optimized** synthesized FA below, this is not the case in the trivial FA's implementation:



Connecting more internal signals to the completion is not possible in the current architecture, as the internal signals are represented as **single rail**, and thus don't acknowledge the finish of the calculation. We will solve this problem in the next subchapter.

2.6.4 Optimized stable adder

As noted in the end of the previous subchapter, the FA must be re-designed in a way that either its **internal signals** will be of dual-rail. It must also output a *done* signal, which would be connected to the completion network of the top adder.

This may be done by using dual-rail *generate* and *propagate* signals [7].

Below is the relevant VHDL code:

```
g(1) <= a(1) and b(1);
p(1) <= (a(1) and b(0)) or (a(0) and b(1));

g(0) <= a(0) or b(0);
p(0) <= (a(0) or b(1)) and (a(1) or b(0));

s_sig(1) <= (p(1) and c_in(0)) or (p(0) and c_in(1));
c_out_sig(1) <= (p(1) and c_in(1)) or g(1);
```

```

s_sig(0)  <= (p(0) or c_in(1)) and (p(1) or c_in(0));
c_out_sig(0) <= (p(0) or c_in(0)) and g(0);

```

```

p_g_done <= p & g;
s_c_done <= s_sig & c_out_sig;

```

```

done_calc_input <= p_g_done & s_c_done;

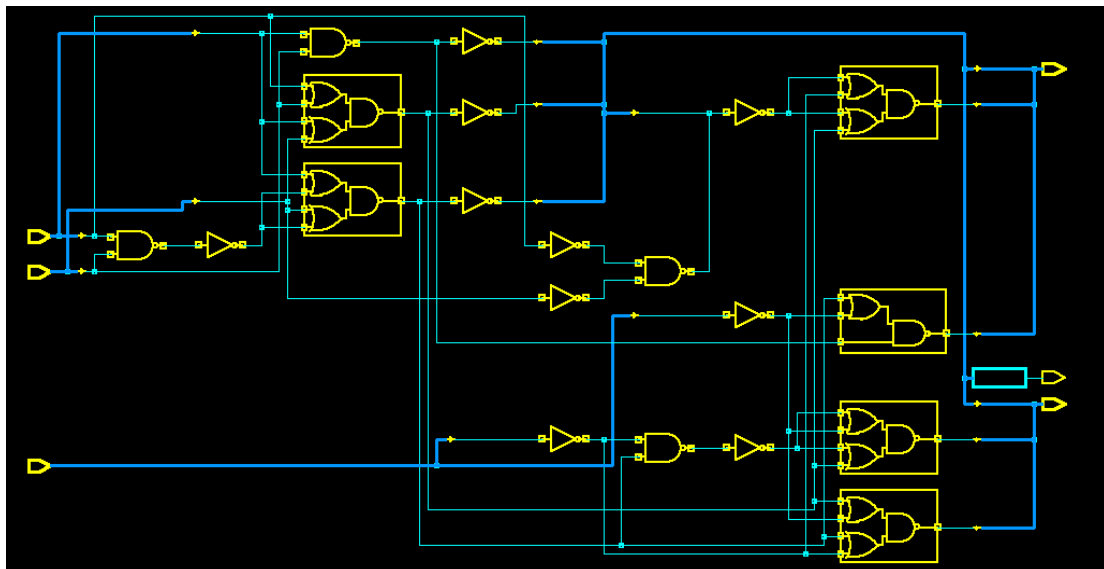
```

```

done_calc : compl_detector
generic map (
  n => 4)
port map (
  input => done_calc_input,
  output => done);

```

Below is the optimized synthesized code. Note, that I assume here that OR2 is a primitive gate, though it's indeed synthesized to NOR2 + INV, as usual in CMOS; similar note holds for the AND2.



Of course, this solution is very pricy in terms of area consumption and delay, due to the large completion network.

2.7.0 The top design

The top entity uses the NCL_X_pipe_lvls for the pipe levels with no combinational logic between each other, and NCL_reg and the NCL_X_adder for the adder and its input registers.

2.7.1 The number of pipe levels

As explained in [5], the minimum number of pipe levels in NCL is 3. This is also a result of the “tokens game”, which is widely discussed in [9]. Thus, the design described in 1.3 contains 3 pipe levels between each virtual time step of the Fibonacci series.

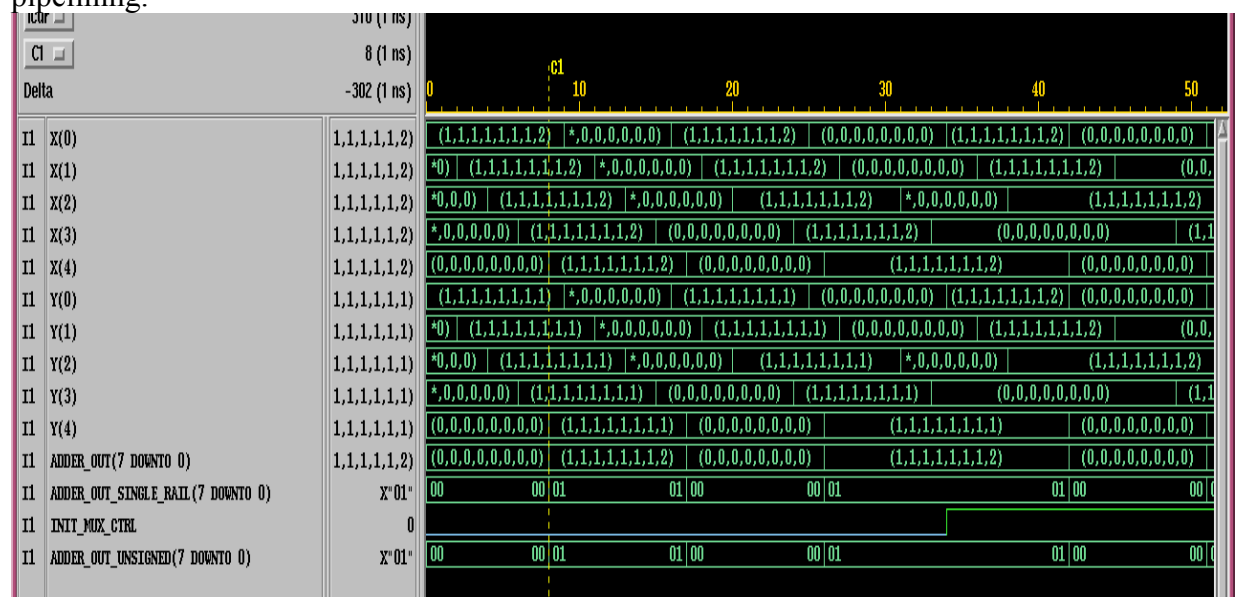
However, as shown in chapter 4 in [9], the optimum point for asynchronous design is usually when the number of DATA tokens equals the number of EMPTY token. In

our case, each virtual time step of the Fibonacci machine would have one DATA token and one EMPTY token; and one additional pipe level between each (DATA, EMPTY) tokens' pair, for holding the **bubble**. In total, **four** pipe levels. Indeed, boosting the TP comes at the cost of additional HW.

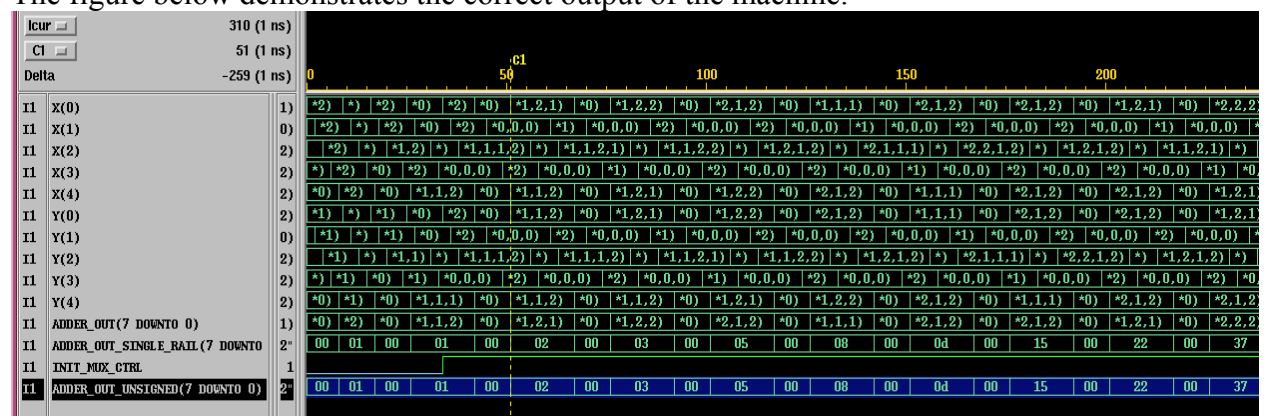
The number of pipe levels is a user-defined parameter in the VHDL code. As the delays of the various units are also parameterized, one may fine-tune his design for finding the best trade-off point between TP and area consumption. (In fact, only 2 options make sense; NUM_OF_PIPE_LVLs = 3; and NUM_OF_PIPE_LVLs = 4. Any value below 3 is forbidden, and any value above 4 is not efficient, as it adds HW with no TP increasing).

2.8.0 Functional verification of the design

In addition to the testbenches of the lower-level blocks, the top Fibonacci series machine was tested. The waveform below demonstrates the correct behavior of the pipelining.



The figure below demonstrates the correct output of the machine.



2.9.0 Synthesis code Vs. Simulation code in NCL_X

NCL_X suggest different HDL coding for simulation and for synthesis. While such a separation is common also in standard synchronous design, as many simulation attributes are not synthesizable (e.g. testbenches, some functions, multi-dimensional

arrays in Verilog etc.), this gap is not desirable, from the obvious reason that we would like the Silicon implementation to be as close as possible to the design which was tested and verified.

Paper review – a comment regarding synthesis in NCL_X

Unclear issue in the suggested NCL_X design flow lies in the suggestion to represent TH gates by Boolean gates, which implement their set functions. While this implementation may give a rough estimation of the area and timing consumption of the circuit, it's still not the desired Silicon implementation of the circuit, as Boolean gates with no memory can't give the control functionality and the DI guarantees which are obtained by using TH gates with hysteresis.

3. Qualitative Comparison: NCL_X Vs. other asynchronous design flows

NCL_X's design flow is indeed quite simple and standard, relatively to other asynchronous design schemes. In this aspect, NCL_X probably outperforms the design scheme used in [4], which extensively use TH mn gates of various values for m , n , thus making the design, simulations and synthesis very complicated. However, one should note that probably an implementation, which is based on TH mn gates is more efficient in terms of timing and area, as such gates offer very efficient transistor-level implementation, and save the large completion network which NCL_X suggests.

In [2] Kondratyev and Lwin use the notation NCL_D referring to a scheme proposed by Lighthart et al. [NCL_D]. Kondratyev and Lwin claim that NCL_X is superior of NCL_D as NCL_X is easier to design, and leaves more room for optimization, due to the explicit completion network which guarantees the DI with no over-designing effort.

This claim indeed makes sense – but only up to a certain level: as demonstrated in 2.6, either when using NCL_X one should be very aware to the synthesis result and not assume that the DI is given automatically by the usage of NCL_X.

In many aspects NCL_X is very similar to the method suggested by David et al. [8] already back in 1992; the latter method will be referred by DRN. Both methods use standard monotonic gates for the implementation of the Boolean functions, and a completion network for the control mechanism. Informally speaking, NCL_X allows concatenation of a few levels of DRN within the same system-level pipeline stage. Theoretically, NCL_X, which is more flexible, leaves more room for optimization. However, as shown in the previous chapter, one should verify that this flexibility doesn't cause a failure to obey the DI requirements, and thus it is not clear whether NCL_X is superior over DRN in practice.

Issue which isn't discussed in many papers is **testability**. Naturally, testing of self-timed circuits is less intuitive and harder to automate than testing of clocked circuits, as the cycle time, and thus either the expected values in the internal signals, are unknown. At this aspect [4] seems to be inferior to the other methodologies discussed above, as it uses general TH mn gates, which don't easily comply to standard testing tools.

4. Conclusions

NCL_X is a simple and straight-forward methodology, which aims to enable the design of asynchronous circuits using standard CAD tools.

Though not a revolutionary break-through, NCL_X is a promising method, which may improve the ability to efficiently design and optimize asynchronous digital circuits.

However, one should use NCL_X carefully for ensuring the delay insensitivity of the lower-level blocks. In addition, the methodology should be more well-defined regarding the employed synthesis coding style.

References

[1] Karl M. Fant and Scott A. Brandt, .NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis,. *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261-273, 1996.

[2] [Design of Asynchronous Circuits by Synchronous CAD Tools](#), Alex Kondratyev , Kelvin Lwin. **IEEE Design & Test**, Volume 19 , Issue 4. pp. 107 – 117. July 2002.

[3] [Design of Asynchronous Circuits by Synchronous CAD Tools](#), Kondratyev, A.; Lwin, K. In proc. Design Automation Conference, 2002. 39th. pp. 411 – 414. June 2002

[4] [Delay-Insensitive Gate-Level Pipelining](#), S.C. Smith, R.F. DeMara, J.S. Yuan, M. Hagedorn, D. Ferguson. *VLSI J.* 30/2 (2001) 103-131.

[5] [Optimization of NULL convention self-timed circuits](#), S. C. Smith, R. F. DeMara, J. S. Yuan, D. Ferguson, D. Lamb. **Integration, the VLSI Journal**, Volume 37, Issue 3, pp. 135 – 165, August 2004.

[6] A CMOS VLSI Implementation of an Asynchronous ALU, in *Asynchronous Design Methodologies*, S. B. Furber and M. D. Edwards, Eds. Amsterdam, The Netherlands: North Holland, 1993, pp. 181–192.

[7] [Delay-Insensitive Carry-Lookahead Adders](#), Fu-Chiung Cheng, Stephen H. Unger, Michael Theobald, and Wen-Chung Cho. In *Proc. Int'l. Conf. VLSI Design*, pp. 322-328. IEEE Computer Society Press, 1997.

[8] [An Efficient Implementation of Boolean Functions as Self-Timed Circuits](#), I. David, R. Ginosar and Michael Yoeli, *IEEE Transactions on Computers*, Vol. 41, No 1, Jan 1992.

[9] Sparsø and Furber, *Principles of Asynchronous Circuit Design*. Kluwer Academic Publishers, c2001

[10] [Asynchronous Design Using Commercial HDL Synthesis Tools](#), Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, Alex Kondratyev. In Proc. of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, April 2000.

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

[HW2] VLSI Architecture course (048878) – HW2 submission, by Itamar Cohen

[HW3] VLSI Architecture course (048878) – HW3 submission, by Itamar Cohen

Appendix A: Clarification mails from the papers' authors.

Alex Kondratyev, one of the authors of the papers [1] and [2], responded a few questions regarding these papers. A copy & paste of this mailing is given below. I would like to thank Mr. Kondratyev for his help.

Mailing regarding [2]

I didn't understand the explanation about figure 2 from the article: It was written there, that the architecture in 2(b) enables a proper acknowledge of y respectively to the inputs a and b. However, as I understand it, if the initial situation was $a=b=0$, and later only b was asserted, the upper AND gate output will be '1', and that will assert the OR3 (which will, in turn, assert y); and if only a was asserted, the middle AND will be asserted - and this will also assert y. So finally, y doesn't acknowledge if only a, only b, or both a and b were asserted - just like in 2(a).

Can you please explain me this issue?

Thanks a lot in advance,

Itamar Cohen,

Technion - Israel Institute of Technology.

Hi Itamar,

You are completely right. The figure is misleading. The point that it should illustrate is that you need to have a non-overlapping cubes in a sum of products to get an acknowledgement property. For that you need to impose the two-phase discipline NULL-DATA and distinguish between direct and inverse values of a signal: for a signal say b you must have b.1 (direct) and b.0 inverse signals. Then the implementation in Figure 2 must be $a.1*b.1+a.0*b.1+b.0*a.1=y$ and this sum of product contains non-overlapping cubes ($a.0 * a.1=0$).

I hope it clears the things.

Thanks

Alex

Hi.

Figure 9: does each of the leftmost gates (e.g., the AND(in1.0, in2.1)) have one output - or two? If it has only one output – I see no difference between the OR threshold gates in the diagram and usual OR2; also, I see no difference between an AND2 threshold gate and a C-element.

And in general – does NCL_X require that the whole INTERNAL signals have dual-rail representation for keeping the DI properties?

You are right, there is no difference between conventional OR2 and OR2 threshold, and threshold AND2 and C-element. This is just the matter of nomenclature. The idea behind the threshold library is to define everything in terms

of threshold functions - some of these threshold functions are well known before. If you mean by internal the signals that are outputs of the gates in a technology mapped circuit which are not observable at the outputs then you are correct. The DI-requirement needs all internals and outputs to be dual-rail.

Mailing regarding [3]

*At section 2 it's written that the proposed method uses a dual-rail encoding, which makes use of DATA and NULL states. Such an encoding fits to the **4-phase** discipline, as two sequencing DATA wavefronts should have NULL separating them.*

However, it's written there that the req-ack mechanism "ensures a two-phase discipline". This makes me a bit confused required the desired discipline for this method.

Thanks in advance.

Hi Itamar,

The confusion arises from different meaning of phases in protocols and datapath. In terms of protocol the suggested approach is 4-phase because it is level based and in every cycle request and ack go up and down. In terms of a circuit behavior it is two phase because every cycle is just DATA and NULL.

Appendix B: Transistor-level implementation of C-element in Verilog.

In contrast to VHDL, Verilog enables transistor-level coding.

Truly, such a code is not synthesizable. Neither does the simulator support keeping of valid values by internal floating nets. Thus, even for testing, THmn hysteresis gates should be coded using a latch process (*always*). Still, the description of a C-element in Verilog is closer to its transistor-level electrical behavior than its VHDL implementation.

Below is the Verilog code:

```
module C_element (c, a, b);
```

```
    output c;
```

```
    input a,b;
```

```
    supply0 GND;
```

```
    supply1 PWR;
```

```
    wire pa_pb;
```

```
    wire na_nb;
```

```
    wire not_out;
```

```
    reg not_c;
```

```
    pmos (pa_pb, PWR, a);
```

```
    pmos (not_out, pa_pb, b);
```

```
    nmos (not_out, na_nb, a);
```

```
    nmos (na_nb, GND, b);
```

```
    always @(posedge not_out) begin
```

```

    if (not_out == 1)
        not_c = not_out;
    end

    always @(negedge not_out) begin
        if (not_out == 0)
            not_c = not_out;
        end

    assign c = ~not_c;

endmodule

```

Appendix C: ReadMe for the code files

The attached VHDL code files are organized as follows:

The code files appear in two different directories: *simulation* and *ams_synthesis*. See within this document details regarding the differences between the two versions.

Files which are of general use within the asynchronous design flow (e.g. C-element) are not specially prefixed. Files which demonstrate a specific NCL (NCL_X) characteristic are prefixed by NCL (NCL_X).

Most internal documentation is found in the **simulation** directory code files. When there's an additional synthesis-specific documentation, it's given within the **ams_synthesis** directory code files.

Compile is the compilation script.

Fibo.cfg is a convenient configuration file for simulation by Synopsys tools.

Fibo.dc is a trivial synthesis script.

Currently the code files are not available on the web, but may be delivered by a personal email.