Asynchronous Design By Conversion: Converting Synchronous Circuits into Asynchronous Ones

Alex Branover, Rakefet Kol and Ran Ginosar
VLSI Systems Research Center, Technion—Israel Institute of Technology, Haifa 32000, Israel

Abstract

A novel methodology and algorithm for the design of large low-power asynchronous systems are described. The system is synthesized by a commercial tool as a synchronous circuit, and subsequently converted into an asynchronous one. The conversion algorithm consists of extracting input and output sets, replacing the storage elements, identifying fork and join sets, and constructing request and acknowledge networks. A DLAP (Doubly Latched Asynchronous Pipeline) architecture is employed. The resulting asynchronous circuit can adapt its effective operating frequency to the supply voltage, facilitating flexible and efficient power management. The algorithm has been validated on several circuits.

1. Introduction

Asynchronous logic has been advocated as a means of reducing power consumption in a number of situations [1-6]. Such circuits typically switch (and consume switching power) only when required or when their inputs change. The power dissipated by the clock tree of a synchronous circuit is eliminated in asynchronous ones. The clock is replaced by local handshake signals, which typically require less power than the clock tree. Since switching power is proportional to the operating frequency, the circuit dissipates less power when the required throughput is reduced. Adaptive supply voltage can be lowered when speed is not required. Since power depends quadratically on voltage, the combination of slow-down and adaptive supply yields a cubic power saving with the reduction of speed. In addition, leakage power, which becomes more significant in newer process technology, can also be managed by reducing the supply voltage. It is easier to vary supply voltage in an asynchronous circuit, since there is no need to coordinate simultaneous variation of the clock frequency.

Unfortunately, achieving such ambitious power savings by asynchronous design has proven to be extremely difficult for designers that are not experts in asynchronous design, because the methodology for the design of large asynchronous logic systems lags substantially behind that of synchronous circuits. Numerous methodologies have been developed for the design of asynchronous systems [7, 8], and a number of special CAD tools for asynchronous systems have been developed. However, there is no tool or methodology for the design of large asynchronous digital systems. The CHP synthesizer [9], TAST [10] and Tangram [11] compile HDL specifications into asynchronous circuits. Such circuits can achieve a number of benefits associated with asynchronous circuits, but they are not necessarily optimized, they require the use of non-standard languages, and are somewhat limited in their applicability or availability. Synthesis tools based on signal transition graphs [12] or burst-mode specifications [13, 14] handle small control circuits and are inappropriate for large digital systems and data processing circuits. A design flow for Globally Asynchronous, Locally Synchronous (GALS) systems on chip (SoC) based on a combination of standard VHDL or Verilog based design of synchronous "islands" with asynchronous "wrappers" has been presented [15] but it implies synchronous blocks and does not support the design of large asynchronous "islands."

This paper proposes to employ standard commercial logic synthesis tools to synthesize a large digital system into a synchronous circuit, and to convert the result into a corresponding asynchronous circuit [16]. The conversion process replaces clocked registers by asynchronous ones and inserts the necessary handshake signals among those registers, without changing the combinational logic. The asynchronous registers result in a *Doubly Latched Asynchronous Pipeline* (DLAP) [17]. This process enables an easy combination of asynchronous blocks with synchronous ones, thus enabling mixed-timed designs and GALS SoCs. This method can also benefit GALS SoCs in simplifying adaptive speed modules: Whereas synchronous modules require adjusting both clock frequency and supply voltage in order to reduce power consumption, converted asynchronous modules automatically adapt their speed and only require voltage adjustment. A somewhat different method, named de-synchronization, has recently been developed [18].

Section 2 presents the DLAP architecture and registers. The conversion algorithm is explained and demonstrated in Section 3 and analyzed in Section 4.

2. DLAP

Large synchronous systems are typically synthesized into deep pipelines, employing a single synchronous clock and edge triggered flip-flops and registers. A similar structure, based on asynchronous registers, is termed a Doubly-Latched Asynchronous Pipeline (DLAP). It operates in a similar manner to its synchronous counterpart, in the sense that all registered may be loaded simultaneously with new data. A DLAP can employ the same combinational logic as the synchronous pipe—an advantage for algorithmic synchronous-to-asynchronous conversion.

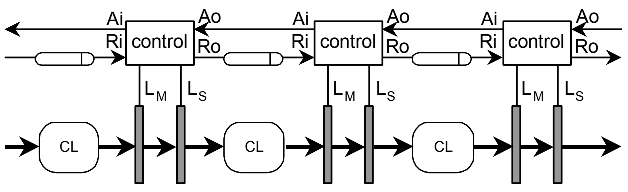

Figure 1: Doubly-Latched Asynchronous Pipeline (DLAP)

DLAP (Figure 1) employs a single rail bundled data and a four-phase handshake protocol. Each stage incorporates two storage elements that resemble the master-slave pair of a synchronous register. If the pipeline is balanced, DLAP operates the same as a synchronous pipeline. At the same time, DLAP retains the benefits of asynchronous pipelines because it is highly decoupled. A pipeline stage that has completed early can start processing the next data even if the following stage is still occupied (the result of the previous computation is safely stored in the master storage element following that stage).

DLAP can be implemented with either edge-triggered registers or transparent latches. Transparent latches are simpler than edge-triggered registers and nearly twice smaller, but since master and slave latches cannot be both open at the same time, the controller is more complex. In this paper we consider only the transparent latch based DLAP.

The signal transition graph (STG) specifying the latch control and the asynchronous control circuit (as synthesized by Petrify [19]) are shown in Figure 2 and Figure 3, respectively. The correct behavior depends on the timing assumptions that assure that all latch control signal transitions are acknowledged. For instance, Dm+ is assumed to arrive at the controller only after Lm+ has propagated through the entire vector of latches that make the master register. This additional safety measure is necessary when the electrical load on signal Lm and hence its delay are unknown in advance and can vary widely, depending on the number of bits per register and actual placement of the latches and routing of the Lm signal.

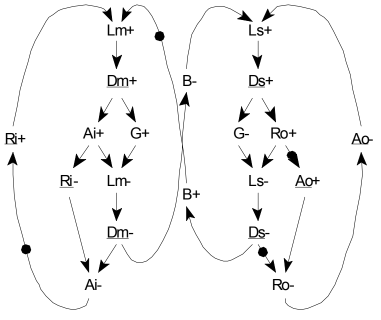

Figure 2: DLAP latch-based controller STG

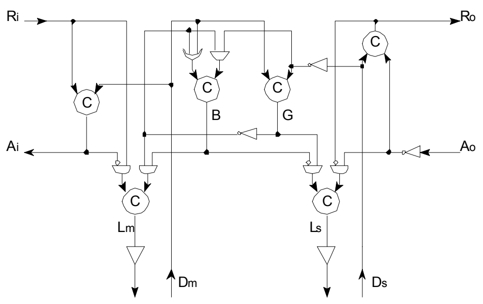

Figure 3: The DLAP master-slave latch control circuit

In addition to regular pipelines, non-linear circuits (e.g., Figure 4) can also be implemented using DLAP, making it the most suitable implementation model for the conversion process.

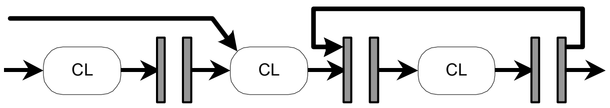

Figure 4: A non-linear circuit

3. The conversion algorithm

Typical logic synthesizers produce a synchronous netlist, comprising combinational logic blocks separated by clocked registers. We convert this conceptual structure as follows. Each register is replaced by a pair of latches and the corresponding asynchronous controller, according to the DLAP design (Section 2). The controllers are interconnected by request and acknowledge handshake signals (we assume that request and acknowledge lines are provided for the external inputs and outputs, respectively). Matched delay lines are inserted on the request lines. The combinational logic blocks are left unchanged. This method, based on single rail / bundled data asynchronous logic, is appropriate for lumped circuits where all delays are well understood so that the timing assumptions associated with bundled data asynchronous design can be assured. The principal advantage of this method is its simplicity and locality (the conversion involves only local transformation—no global redesign of the circuit is required). For distributed systems (such as spanning wide areas of a large SoC), other conversion methods, e.g. based on dual-rail or 1-of-4 signaling, may be more appropriate.

The following sub-sections describe the steps of the conversion algorithm.

3.1. Input/Output extraction

At first the algorithm examines all registers of the netlist and identifies all *inputs* to and *outputs* from combinational logic blocks. For example, the synchronous circuit shown in Figure 5 incorporates three flip-flops and three combinational logic blocks. Inputs into combinational blocks are listed in the *inputs* structure, which combines two sets, *ExternalInputs* (external inputs to the net) and *FlipFlopOutputs*. In Figure 5, *ExternalInputs* = {I1}, *FlipFlopOutputs* = {I2, I3, I4}, and *inputs* = {I1, I2, I3, I4}. The *outputs* structure also combines two sets, *ExternalOutputs* (O4 in Figure 5) and *FlipFlopInputs* ({O1, O2, O3}). Thus, *outputs* = {O1, O2, O3, O4}.

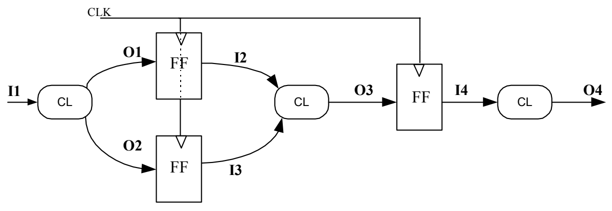

Figure 5: A synchronous circuit

3.2. Register and flip-flop replacement

At this second step, the algorithm identifies all flip-flops or registers. Each flip-flop is replaced by a DLAP single-bit register and control logic. Only one control logic block is required for all the bits of the same register. Figure 6 shows the result of this step applied to the example circuit of Figure 5.

The four handshake signals of each controller (RI, AI, RO, AO) are interconnected to other controllers during the steps below that construct the Request and Acknowledge networks. But before they can be interconnected, Fork and Join sets must be generated.

3.3. Creating the Fork and Join sets

A *Fork* set, containing elements of the *outputs* structure, is associated with each element of the *inputs* structure. Output Oi∈*outputs* is a member of the *Fork* set of input Ik∈*inputs* if a directed combinational path exists from Ik to Oi. Using the full timing graph of the original synchronous circuit, it is straightforward to check the existence of such combinational paths. The path is considered existing even if *delay*(Ik ,Oi)= 0 . Obviously, *Fork*(Ik) is the successor set of Ik.

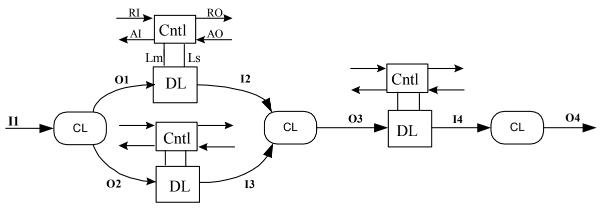

Figure 6: Circuit after flip-flop replacement

Similarly, a *Join* set is created for each element of the *outputs* structure, but only elements from the *inputs* structure could be included in the *Join* set. The indication of the membership of some input Im in the *Join* set of the output Ot is the same as for *Fork* set, namely a combinational path from Im to Ot exists in the timing file. In other words, *Join*(Ot) is the predecessor set of Ot. For instance, the combinational logic circuit of Figure 7 incorporates two inputs (I1,I2) and two outputs (O1,O2), and all the combinational paths from inputs to outputs are marked by dashed lines. Applying the Fork/Join definitions to this circuit, the following sets are created: *Fork*(I1)={O1} ; *Fork*(I2) = {O1 ,O2}; *Join*(O1) = {I1 , I2}; *Join*(O2) = {I2}.

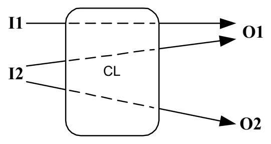

Figure 7: A 2-input, 2-output combinational logic block

Likewise, the *Fork* and *Join* sets of the example circuit of Figure 5 are *Fork*(I1)={O1,O2}; *Fork*(I2)={O3}; *Fork*(I3)={O3}; *Fork*(I4)={O4}; *Join*(O1)={I1}; *Join*(O2)={I2}; *Join*(O3)={I2,I3}; and *Join*(O4)={I4}.

3.4. Constructing the Request network

This step produces all the request signals to all the controllers, and inserts the required matched delays and C-elements. Both *Fork* and *Join* sets created during the previous step are used in constructing the Request network.

The elements of the structure *inputs* (I1, I2, I3 and I4 in the example of Figure 5) constitute the starting points for all combinational paths in the circuit. Hence, a request signal indicating valid data on Ik ∈ *inputs* should be sent to the control logic of all stages whose inputs are members of *Fork*(Ik). Consider the first *Fork* set of Figure 5, *Fork*(I1)={O1,O2}. I1 is a starting point of two combinational paths, leading to O1 and O2. The respective request lines are labeled req_I1, req_O1 and req_O2. The combinational paths connecting I1 with O1 and O2 imply that req_I1 must be connected with req_O1 and req_O2 (Figure 8). Next, we consider the computational delays from I1 to O1 and O2: d11=*delay*(I1,O1) and d12=*delay*(I1,O2). The *delay*() function always produces an upper bound on the combinational delay plus a safety margin, and that value is used as the matched delay that is inserted on the request line, as in Figure 9. Formally:

- For each pair of (input, output) nodes (Ik, Oi), if Oi∈*Fork*(Ik) then a request line is introduced from Ik to Oi.
- A matched delay of *delay*(Ik,Oi) is inserted onto the (Ik, Oi) request line.

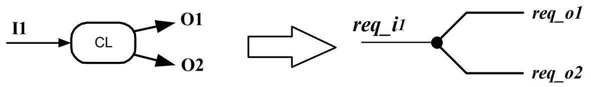

Figure 8: *Fork*(I1)

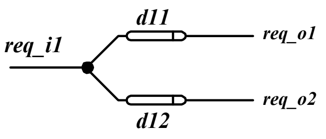

Figure 9: *Fork*(I1)—Request network with delays

The Request network in Figure 9 duplicates the delay lines. It can be optimized as follows, resulting in reduced area and power:

Find the shortest delay d_{min} among all delays on the request lines emanating from Ik.

- Introduce a d_{min} delay line onto the request line starting at req_Ik and ending at the fork node that forks req_Ik to all its destinations.
- For each request line leading from the fork node to a destination Oi, subtract d_{min} from the delay on that request line.

Figure 10 shows the above example after optimization, assuming that d11 < d12. A complementary process is now applied to the *Join* sets. For instance, consider *Join*(O3)={I2, I3}. A request line is generated for each pair, (I2,O3) and (I3,O3), as in Figure 11. The two request lines are combined with a C-element, which serves as an "event AND" gate (Figure 12). Formally,

- For each pair of (input, output) nodes (Ik, Oi), if Ik∈*Join*(Oi) then a request line is introduced from Ik to Oi.
- A matched delay of *delay*(Ik,Oi) is inserted onto the (Ik, Oi) request line.
- If *n*=‖*Join*(Oi)‖ > 1 then an *n*-input C-element is employed to combine all *n* request lines converging onto Oi. The output of that C-element is req_Oi.

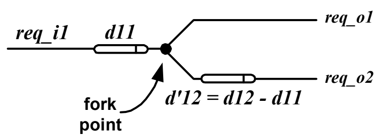

Figure 10: *Fork*(I1)—Optimized Request network

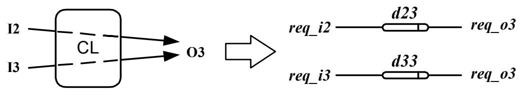

Figure 11: *Fork*(I2) and *Fork*(I3) request lines

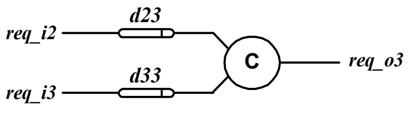

Figure 12: *Fork*(I2), *Fork*(I3) and *Join*(O3) Request network

The situation with O1 and O2 is simpler because their *Join* sets contain only one member, I1. In fact, a C-element is not required in this case. Both O1 and O2 are combinational functions of only I1 and the Request network constructed by the Fork step (Figure 9) is sufficient. The Request network for the (I4,O4) path is trivial, containing neither fork nor join. The resulting example circuit after constructing the Request network is shown in Figure 13.

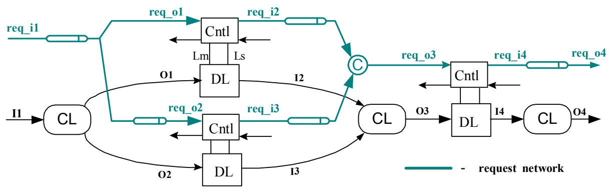

Figure 13: The circuit with the Request network

3.5. Constructing the Acknowledge network

In contrast with the Request network, the construction of the Acknowledge network is based only on the Fork set. Input Ik∈*inputs* is connected by a combinational path with each output Oi∈*Fork*(Ik). As we have already noticed, Ik is an input to the combinational logic block and at the same time it is an output of some previous stage (or an external input). On the other hand, Oi is an output of the combinational logic block and an input to the following stage (or an external output). The stage where Ik is output is permitted to issue valid data only after all the succeeding stages have signaled their readiness to accept it. In other words, Ik should be acknowledged by every stage input Oi∈*Fork*(Ik). This implies the use of an *m*-input C-element per each Ik, where *m*=||*Fork*(Ik)||. Formally, the Acknowledge network is produced as follows:

- For each pair of (input, output) nodes (Ik, Oi), if Oi∈*Fork*(Ik) then an acknowledge line is introduced from Oi to Ik.
- If *n*=||*Fork*(Oi)|| > 1 then an *n*-input C-element is employed to combine all *n* acknowledge lines converging onto Ik. The output of that C-element is ack_Ik.

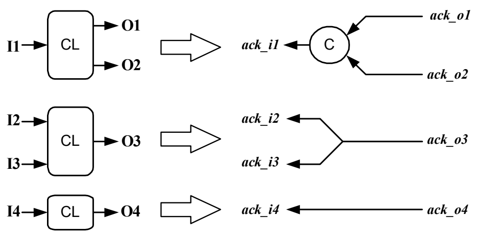

Figure 14: The Acknowledge network

The portions included in the Acknowledge network of our original circuit are shown in Figure 14, and Figure 15 shows the complete circuit after all acknowledge lines are inserted.

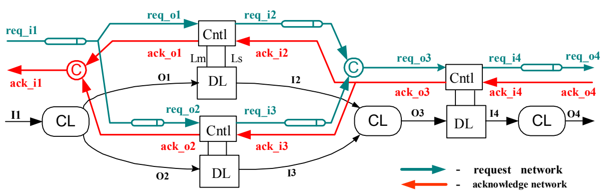

Figure 15: The final converted circuit

3.6. Bus processing

Bundled multi-bit data buses are treated somewhat differently than the foregoing steps, which are related to single flip-flops. Consider the *n*-bit buses O[n], I[n] of the synchronous circuit in Figure 16.

In the converted circuit (Figure 17), the asynchronous register contains 2*n* latches (a master and a slave for each bit) and a single controller, producing a single Lm and a single Ls latch-enable signals for the entire register. The single request line req_o' signals validity of all *m* bits of the O' bus. The matched delay d' accounts for the worst case delay over the combinational logic over all *n* bits of the O[n] bus. Consequently, req_o implies validity of the entire O[n] bus:

d'=max(delay(O'[i], O[j])

{ i=l...m, j=1...n, where the (O'[i], O[j]) path exists }

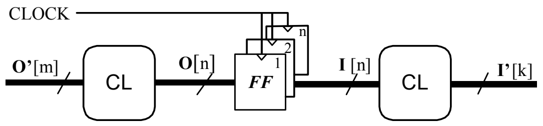

Figure 16: A synchronous n-bit bus

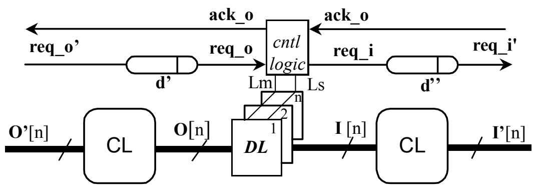

Figure 17: A converted asynchronous n-bit bus

This approach is independent of the nature of the combinational logic and saves on hardware, but it requires a search of maximal delay and results in worst-case delays.

4. Analysis

Two of the algorithm steps, the creation of *Fork* and *Join* sets and the construction of Request and Acknowledge networks, incur O(K^2) time complexity, where K is the maximum of the number of external outputs, the number of external inputs, and the number of flip-flops, rendering the entire algorithm asymptotically quadratic in execution time.

A number of synchronous circuit examples, incorporating a variety of flip-flops and combinational elements, have been synthesized using Synopsys and the converted into asynchronous circuits (Table 1). Control logic for double-latches required 23 gates in this design and constituted the principal contribution to increasing the circuit area. The circuits that contained many single flip-flops (rather than buses) where each flip-flop is replaced by a double-latch plus control logic incurred a larger increase than circuits with buses.

Area overhead diminishes as the original synchronous circuit grows (Figure 18). In our larger circuits, where the area overhead contributed by DLAP control logic and delay lines was about 26% of the combinational logic area, the clock network that was eliminated in the process of conversion had occupied approximately 10% of the original circuit area. As a result, the converted asynchronous circuit was about 16% larger than the original synchronous one.

5. Conclusions

Synchronous-to-asynchronous conversion enables easier transition to asynchronous design and at the same time retains investment in existing synchronous tools and methodologies. In addition it enables asynchronous interface to other asynchronous and synchronous modules (mixed timing). Asynchronous low power techniques, (such as variable power supply) can be adopted for the generated asynchronous structure. Reducing the supply voltage is extremely effective in saving switching power whenever the circuit can be operated at a slower speed, and it is also useful for mitigating power loss due to leakage. The main cost of asynchronous conversion is area increase, becoming relatively smaller for larger circuits.

Table 1: Example of Circuit Conversions

Ckt	Function	Gates in Sync Ckt	Added Control Logic Units	Added Delay Gates	Added Gates in Async Ckt	Total Gates in Async Ckt	Area Growth
N1	Test Circuit	18	1	14	37	55	206%
N2	Test Circuit	32	2	10	56	88	175%
N3	Error Detector	113	5	32	147	260	130%
N4	Traffic Light Controller	64	1	48	71	135	111%
N5	Data Path	780	7	74	210	990	27%
N6	FIR Filter	1148	10	93	293	1441	26%

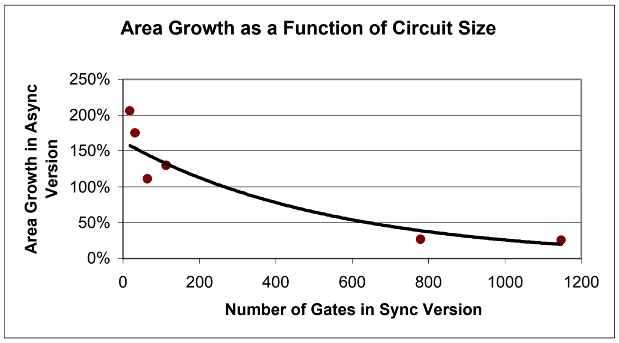

Figure 18: Area Growth. Converted circuits eliminate the clock tree, but require additional area for control and delay buffers.

The sync-to-async conversion algorithm has been presented. Run-time complexity is quadratic in the size of the input. The algorithm was implemented as a CAD tool and tested and proven on different circuits. DLAP (doubly-latched asynchronous pipeline), which imitates synchronous pipes, is used as a target structure of the conversion algorithm. The method exploits an existing trusted synchronous synthesizer (Synopsys).

Further research may be targeted at improving run-time complexity by taking advantage of the synthesizer's internal timing data. The algorithm may also be extended to handle high-performance dynamic logic (domino, self-reset domino) and non-pipelined structures (memories, caches). Converting into other types of asynchronous circuits, such as micropipelines or quasi-delay-insensitive circuits, may also be addressed by future research.

References

[1] Berkel *et al.*, "Asynchronous Circuits for Low Power," *IEEE Design & Test of Computers*, 11, 22-32, 1994.

[2] Nielsen *et al.*, "Low-power operation," *IEEE Trans. VLSI*, 2, 391-397, 1994.

[3] Piguet, "Low-power and low-voltage CMOS digital design," *Microelect. Eng.,* 39, 179—208, 1997.

[4] Berkel *et al.*, "Asynchronous Does Not Imply Low Power, But ..." *Low Power CMOS Design*, Chandrakasan and Brodersen (ed), 227—232, 1998.

[5] Martin, "Remarks on low-power advantages of asynchronous circuits," *ESSCIRC,* 1998.

[6] Kessels and Peeters, "The Tangram Framework," *ASP-DAC*, 255—260, 2001.

[7] Hauck, "Asynchronous Design Methodologies," *Proc. IEEE*, 83, 69—93, 1995.

[8] Nowick *et al.*, "Asynchronous Circuits and Systems," *Proc. IEEE*, 87, 219—222, 1999.

[9] Martin, "Programming in VLSI," *Concurrency and Communication,* Hoare (ed), 1—64, 1990.

[10] Renaudin *et al.*, "A Design Framework for Asynchronous/Synchronous Circuits Based on CHP to HDL Translation," *ASYNC*, 135—144, 1999.

[11] Berkel *et al.*, "VLSI Programming and Silicon Compilation," *ICCD*, 150—166, 1988.

[12] Cortadella *et al.*, "Designing Asynchronous Circuits from Behavioral Specifications with Internal Conflicts," *ASYNC*, 106—115, 1994.

[13] Yun and Dill, "Automatic Synthesis of 3D Asynchronous State Machines," *ICCAD*, 576—580, 1992.

[14] Fuhrer *et al.*, "Minimalist," Columbia University, 1999.

[15] Oetiker *et al.*, "Design Flow for a 3-Million Transistor GALS Test Chip," *ACiD Workshop*, 2003.

[16] Kol, Ginosar and Samuel, "Statechart Methodology for Asynchronous Systems," *ASYNC*, 1996.

[17] Kol and Ginosar, "A Doubly-Latched Asynchronous Pipeline," *ICCD*, 706—711, 1997.

[18] Cortadella *et al.*, "A Concurrent Model for De-Synchronization," *IWLS*, 2003.

[19] Cortadella *et al.*, "Petrify," *IEICE Trans. Information and Systems*, E80-D, 315—325, 1997.