Asynchronous Design—Part 2: Systems and Methodologies

Steven M. Nowick
Columbia University

Montek Singh
University of North Carolina at Chapel Hill

Editor's notes:
The second part of the two-part tutorial on asynchronous design addresses methodologies for designing asynchronous systems and CAD tool flows. It also presents a summary of asynchronous processors and architecture, as well as testing.

—Partha Pratim Pande, Washington State University

■ **This two-part** article aims to provide both a short historical and technical overview of asynchronous design, as well as a snapshot of the state of the art. Part 1 covered foundations of asynchronous design, and highlighted recent applications, including commercial advances and use in emerging application areas. Part 2 focuses on methodologies for designing asynchronous systems, including basics of hazards, synthesis and optimization methods for both logic-level and high-level synthesis, and the development of specification languages and CAD tool flows. Finally, two sidebars provide a summary of asynchronous processors and architectures, as well as testing.

Synthesis and optimization

The synthesis of asynchronous circuits poses some unique challenges, given the absence of a fixed-rate global clock. There is a large body of research to develop sound and efficient techniques, both for logic-level and high-level synthesis.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.
Digital Object Identifier 10.1109/MDAT.2015.2413757
Date of publication: 16 March 2015; date of current version: 04 June 2015.

Hazards and logic synthesis

At the logic level, the lack of a fixed-rate clock implies an event-driven paradigm, where components are activated and compute whenever they receive new inputs. As a result, without a validating clock edge, each component must clearly signal when it has produced valid data, and therefore—unlike synchronous design— glitches must be avoided.

The potential for a glitch is called a hazard [1]. The goal of asynchronous logic synthesis is to provide optimal implementations that are also hazard-free.

Hazards are temporal phenomena, that are defined with respect to an *input transition*, where one or more input signals change value. A *static logic hazard* occurs when an output is meant to remain stable (i.e., at 0 or 1), but instead may glitch. A *dynamic logic hazard* occurs when an output is meant to have a clean transition (i.e., $0 \rightarrow 1$ or $1 \rightarrow 0$), but instead may glitch.

Figure 1a illustrates a dynamic logic hazard for a given input transition, where the circuit output has a function transition, in this case from $1 \rightarrow 0$. Depending on the relative arrival times of input A and C transitions, and the gate delays, the output may glitch, as shown. The circuit in Figure 1b has identical functionality but is guaranteed hazard-free for this input transition: with a small modification to the bottom AND gate, it provides a clean $1 \rightarrow 0$ output transition regardless of gate and wire delays.

The fundamental challenge of asynchronous logic synthesis is to develop optimization techniques

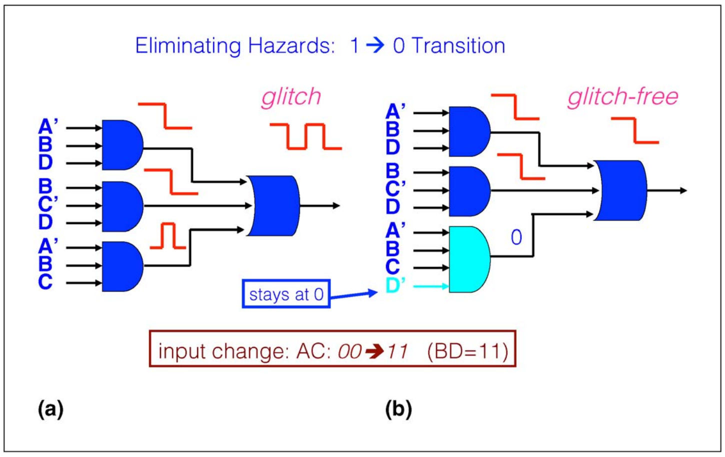
Eliminating Hazards: 1 → 0 Transition
glitch
glitch-free
A'
B
D
B
C'
D
A'
B
C
A'
B
D
B
C'
D
A'
B
C
D'
0
stays at 0
input change: AC: 00→11 (BD=11)
(a)
(b)

Figure 1. Example of a dynamic logic hazard.

at each level of the classic synthesis flow—two-level logic minimization (i.e., targeting AND/OR or NAND/NAND structures), multi-level optimization, optimal technology mapping to cell libraries—which simultaneously guarantee hazard-freedom.

Early work on hazards and hazard-free design was developed by Huffman, Unger, McCluskey, Eichelberger and Muller in the 1950s and 1960s, much of which targeted correctness under single-input transitions or highly restricted multiple-input transitions. An exact solution to the two-level hazard-free logic minimization problem was proposed in the early 1990s [1]. Techniques have also been introduced for heuristic two-level hazard-free logic minimization [2], multi-level hazard-free logic minimization [3] (by identifying safe "hazard-non-increasing" transformations), and hazard-free technology mapping. A general methodology has been proposed, which identifies, for a given arbitrary Boolean function and set of specified input transitions, whether any hazard-free multi-level implementation exists [4].

Two alternative widely-used approaches for the specification and synthesis of hazard-free asynchronous controllers have been proposed: 1) burst-mode (BM) and 2) Petri-net based.

A burst-mode specification, shown in Figure 2a, is a form of an asynchronous Mealy machine [5]. Unlike synchronous specifications, state transitions occur without the notion of clock cycles, but rather are event-driven. Once a specified "input burst" of one or more signal transitions arrives, in arbitrary order and time, the specified output changes (i.e., "output burst") are generated, and the machine advances to the next specification state. Burst-mode specifications were first defined and formalized by Nowick et al., including the first guaranteed hazard-free gate-level synthesis flow [5]. The method built on an earlier more *ad hoc* approach which was used in the design of a large-scale experimental communication coprocessor (see Post Office chip, in "Processors and Architecture" sidebar). A latchless architecture is targeted [2], which facilitates low latency, consisting of standard-cell-based combinational logic, and with the state stored on fed-back outputs. (A latch-based architecture was also proposed [5].) A simple hold-time requirement must be met: no input burst may arrive until the machine has stabilized from the previous input burst. This so-called ''*generalized fundamental mode*'' timing requirement, when handling a multi-input burst, is an extension of classic ''*fundamental mode*'' which assumed only single-input changes [2], [5]. Burst-mode controllers have been used in a number of applications, including: an experimental low-power infrared communication chip [6] (see example in Figure 2a) and high-performance Post Office communications coprocessor, both from HP Laboratories; control units in the Intel RAPPID project (see Part 1), which includes the use of "extended burst-mode (XBM)" specifications by Yun and Dill; and an experimental laser space measurement chip developed at NASA Goddard.

Petri-net based controller design has also been widely used, especially as supported in the *Petrify* synthesis flow [7]. Figure 2b illustrates a concurrent controller specification using a State Transition Diagram (STG), a restricted form of Petri net, for a VME bus receiver. This versatile style allows highly concurrent control and interface circuits to be easily specified, with synchronization points and concurrent threads captured explicitly. A quasi-delay insensitive (QDI) timing assumption is used, which requires that, at each wire fanout point in the circuit, each wire fork has roughly equal delay. Beyond this "isochronic fork" assumption, there are no environmental hold-time requirements. This approach grew out of early work by Muller and Varshavsky. The methodology has had wide application, including for a concurrent VME bus controller (see Figure 2b), and for several control circuits in asynchronous ARM processors from the University of Manchester (Amulet3 and 3i, see "Processors and Architecture" sidebar).

While burst-mode and Petri-net-based methods can often be used to synthesize the same specifications, they build on two distinct views. A BM approach, while

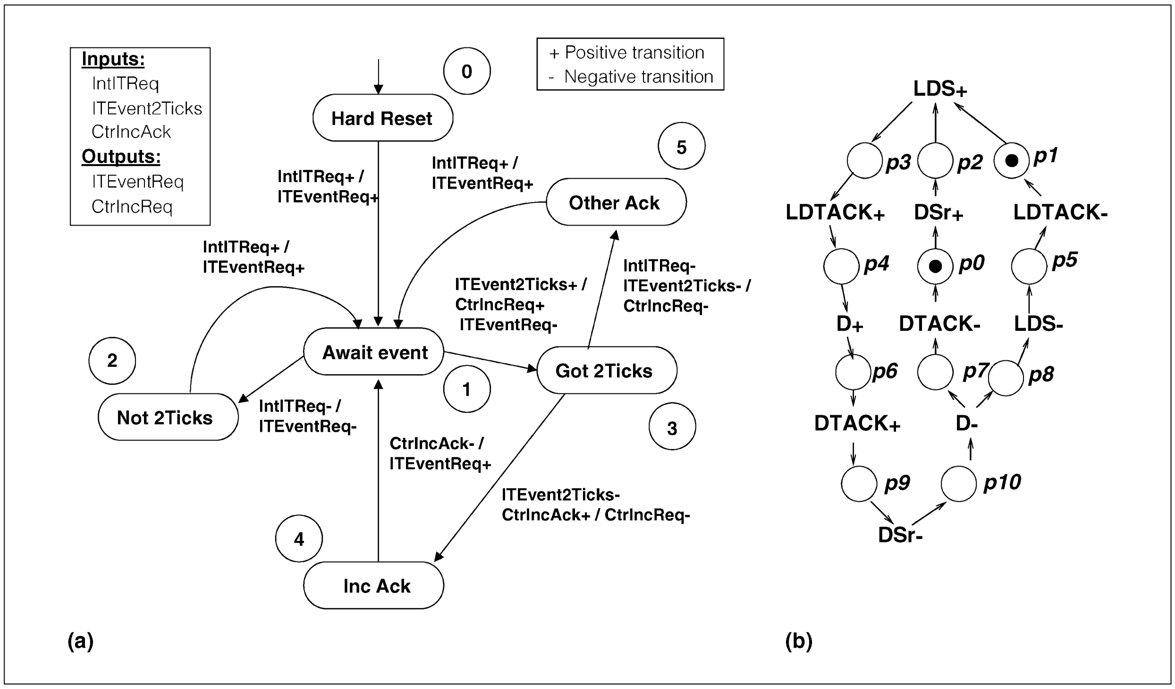

Figure 2. (a) Burst-mode specification and (b) Petri net specification.

allowing significant concurrency in specifications, is fundamentally state-based, as illustrated in Figure 2a, and typically assumes some moderate hold-time environmental timing assumptions. These restrictions allow the development of highly-efficient algorithms [2], [5], which leverage existing synchronous approaches, including exact and heuristic solutions for optimal state assignment and logic minimization, as well as support for handling larger examples [6]. As such, they are especially suitable for high-performance controller design [2], [6] (see also Post Office chip in "Processors and Architecture" sidebar). A Petri-net-based approach fundamentally aims at supporting fine-grain concurrency in specifications, as illustrated in Figure 2b, and typically targets QDI circuits with no environmental timing assumptions. These techniques do not have direct synchronous antecedents, therefore sophisticated algorithms have been developed to ensure correct state assignment and logic optimization [7]. As such, they are especially suitable for small highly concurrent interface circuits. However, beyond these original center points, each approach has been subsequently broadened, with support for increased concurrency in burst-mode methods using XBM, and to allow support for relative timing and burst-mode assumptions in Petri-net methods.

An alternative approach, called *NULL Convention Logic (NCL)*, introduced by Karl Fant, has been proposed for the unified synthesis of both control and datapath. In NCL, synchronous netlists are translated directly to equivalent asynchronous dual-rail circuits using threshold gates, such as m-of-n cells [8]. Entire asynchronous systems can be designed which are highly robust to process, temperature and voltage (PVT) variability as well as extreme environmental conditions (see Part 1), using a variant of QDI design; however, the circuits tend to have large latency and area overheads due to their dual-rail encoding and complex threshold networks. Several recent optimization strategies have been proposed to significantly reduce these overheads (see "Specification languages and tools" section, below).

Finally, several approaches have been proposed for timed circuits, which exploit knowledge of environmental and internal delays to significantly optimize designs. A general framework, called "relative timing [9]," due to Stevens, Ginosar and Rotem, supports partial ordering requirements between different paths, and was used in the Intel RAPPID project (see Part 1). An alternative approach by Myers and Meng exploits absolute timing information [10].

High-level synthesis

High-level synthesis is the automated conversion of an algorithmic description of the behavior of a system into a structural representation. Typically, the system's behavior is specified in a hardware description language (HDL), and then automated tools explore the design space according to cost functions and constraints, resulting in an efficient gate-level implementation that can be handed off to physical design tools.

An early asynchronous approach, called ACK, was developed at the University of Utah [11]. A system is specified at a procedural level using Standard Verilog HDL, with an add-on package of asynchronous channel abstractions, and the compiler maps it to distributed asynchronous control and datapath blocks. Three controller types are supported: burst-mode [2], macromodular [12], and microprogrammed control. The compiler implements several automated optimizations, including control partitioning, loop unrolling, sharing of common subexpressions and dead-code elimination. Transformations are back-annotated in the original specification to provide useful designer feedback. The approach was applied to synthesize a complete error decoder for a CD player. Overall, this framework was an important early contribution to the automated synthesis of asynchronous systems, but lacked the ability to optimally target cost functions.

A classic high-level synthesis problem is *resource sharing*, which aims to determine how a limited set of system resources (ALUs, registers, memories, etc.) can be optimally shared and scheduled among the specified operations so as to minimize a cost function (e.g., latency) under cost constraints (e.g., area). While much work has been done on synchronous resource sharing, the problem is fundamentally different for asynchronous systems. In particular, synchronous resource sharing methods typically schedule all operations at the beginning of clock intervals, yielding a discrete formulation that is often cast as an integer linear programming problem. An asynchronous system, on the other hand, has a continuous time scale: an operation can be scheduled as soon as its operands are available and a resource is ready. An optimal asynchronous schedule cannot be obtained from a synchronous formulation merely through a post-processing step of relaxing the clocking boundaries [13]. Nonetheless, some early asynchronous resource sharing approaches directly adapted the synchronous model, though results were not guaranteed to be optimal.

A truly asynchronous approach must abandon the discrete-time formulation and, instead, must consider the problem as one of determining a *partial ordering of operations*. Such a formulation was first proposed by Badia and Cortadella [14] using a heuristic list-scheduling approach, where, at each step, the next operation to be scheduled is chosen in a greedy manner, i.e., the one with the longest chain of dependent operations. The approach only handled resource constraints, it did not consider latency constraints. The first exact approach to asynchronous resource sharing was introduced by Hansen and Singh [13], using a branch-and-bound strategy, which also supported sophisticated multi-objective cost functions and constraints, e.g., resource- or area-constrained latency minimization, and latency-constrained area minimization. Subsequent recent extensions and generalizations of this work incorporate energy and power constraints, as well as scalability through a hierarchical partitioning and scheduling approach.

A different flavor of the scheduling problem, where asynchronous approaches have surpassed synchronous ones, is one of optimal scheduling of *pipelined loops*, i.e., where multiple iterations of an algorithmic loop can execute in an overlapped fashion, called *multi-token execution*. The goal of this problem is to minimize the cycle time of the entire loop execution, instead of the latency of one loop iteration. The first exact solution, whether synchronous or asynchronous, was introduced by Hansen and Singh [15]. This problem has long been a challenge even in synchronous design, and is much harder than the basic scheduling problem, because one cycle of the schedule can mix-and-match operations from an arbitrary number of successive loop iterations.

Specification languages and tool flows

A key component in facilitating the practical use of asynchronous design is to provide entry hardware description languages (HDLs) and automated computer-aided design (CAD) tool flows. There have been important advances on both fronts, though more remains to be done.

One of the interesting research challenges has been to resolve the balance between two competing needs: 1) to provide compatibility with existing synchronous specification languages and CAD tool flows, and 2) to design specification languages that best capture the fine-grain concurrency, distributed synchronization, and underlying clockless paradigm of asynchronous systems. For the former approach, there has been a strong push to demonstrate that

Processors and Architecture

Several leading processors from the 1950's to 1970's used asynchronous circuits extensively, including the ILLIAC [1952] and ILLIAC II [1962] (University of Illinois); the Atlas [1962] and MU-5 [1974] (University of Manchester); the DDM-1 dataflow machine[1] [1976] (Al Davis/Burroughs); and designs from the seminal Macromodules project,[2] which enabled the rapid plug-and-play construction of custom systems using pre-existing building blocks [mid 1960's] (Washington University).

The first modern single-chip asynchronous microprocessor was designed by Martin's group at Caltech [1988].[3] This 16-bit RISC processor was developed as a proof-of-concept to demonstrate the CHP compilation approach, and the speed and robustness of QDI circuits. Subsequently, the Amulet1 microprocessor, first in a series, was developed at the University of Manchester by Furber's group [1993] as an asynchronous ARM using micropipelines. Both projects were highly influential, setting a foundation for two decades of technical advances in all aspects of architecture, including pipeline circuits, cache and memory design, speculation, exception handling, and on-chip networks. These first forays were quickly followed by more advanced designs: an 8-bit MIPS R2000-like TITAC-1 [1994] (Nanya's group, Tokyo Institute of Technology); Amulet2e [1997]; TITAC-2[4] [1997], which features a 32-bit MIPS R2000 microprocessor using a novel 'scalable delay-insensitive' circuit model, allowing parameterized robustness requirements; a full-featured 32bit asynchronous R3000 microprocessor called MiniMIPS[3] [1997] (Martin's group, Caltech); and Amulet3i[5] [2000]. The Amulet3i was a true system-on-chip, organized modularly around a flexible asynchronous on-chip bus, and extensible through addition of conventional clocked IP.

Several low-power microcontrollers have also been designed. The asynchronous 8051 microcontroller from Philips[6] not only was a major commercial success, selling over 700 million copies, but also used the first fully-automated industrial-strength asynchronous design flow (Tangram, later Haste). More recently, Caltech's Lutonium has demonstrated ultra-low-energy operation at less than a nanojoule per instruction.[3]

Other important milestones, using novel architectures, include the counterflow pipeline processor at Sun Microsystems Laboratories,[7] an asynchronous out-of-order architecture featuring precise exceptions at University of Utah,[8] a super-pipelined multimedia processor at Sharp,[9] the Post Office communication co-processor at HP Laboratories,[10] and a low-power sensor-network processor from Cornell University.[11] Asynchronous design has also been used as a foundation for large-scale inter-processor communication, including the Torus routing chip,[12] FLEETzero at Sun Microsystems Laboratories,[13] and the terabit-rate commercial crossbar switches of Intel/Fulcrum.[14] Finally, the recent surge of interest in cognitive computing is exemplified by several recent neuromorphic processors — IBM's TrueNorth,[15] Stanford University's Neurogrid (Boahen's group),[16] and University of Manchester's SpiNNaker[17] (Furber's group) — all of which use fully-asynchronous interconnection networks to integrate massively-parallel architectures with thousands —and, eventually, millions— of processing elements.

References

1. A. L. Davis, "The architecture and system method of DDM1: A recursively structured data driven machine," in *Proc. 5th ACM ISCA*, 1978, pp. 210–215.
2. W. A. Clark and C. E. Molnar, "Macromodular computer systems," in *Computers in Biomedical Research*, vol. 4, R. Stacy and B. Waxman, Eds. New York, NY, USA: Academic, 1974, pp. 45–85.
3. A. J. Martin, M. Nystrom, and C. G. Wong, "Three generations of asynchronous microprocessors," *IEEE Des. Test*, vol. 20, no. 6, pp. 9–17, 2003.
4. A. Takamura et al., "TITAC-2: An asynchronous 32-Bit microprocessor based on scalable-delay-insensitive model," in *Proc. ICCD*, 1997, pp. 288–294.
5. J. D. Garside et al., "AMULET3i—An asynchronous system-on-chip," in *Proc. Int. Symp. Adv. Res. ASYNC*, 2000, pp. 162–175.
6. H. van Gageldonk et al., "An asynchronous low-power 80C51 microcontroller," in *Proc. Int. Symp. Adv. Res. ASYNC*, 1998, pp. 96–107.
7. R. F. Sproull, I. E. Sutherland, and C. E. Molnar, "The counterflow pipeline processor architecture," *IEEE Des. Test*, vol. 11, no. 3, pp. 48–59, 1994.
8. W. F. Richardson and E. Brunvand, "Precise exception handling for a self-timed processor," in *Proc. ICCD*, 1995, pp. 32–37.
9. H. Terada, S. Miyata, and M. Iwata, "DDMPs: Self-timed super-pipelined data-driven multimedia processors," in *Proc. IEEE*, vol. 87, no. 2, pp. 282–295, 1999.
10. B. Coates, A. L. Davis, and K. Stevens, "The post office experience: Designing a large asynchronous chip," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 3, pp. 341–366, 1993.
11. V. Ekanayake et al., "An ultra low power processor for sensor networks," in *Proc. Int. Conf. ASPLOS*, 2004.
12. W. J. Dally and C. L. Seitz, "The torus routing chip," *Distrib. Comput.*, vol. 1, no. 4, pp. 187–196, 1986.
13. W. S. Coates et al., "FLEETzero: An asynchronous switching experiment," in *Proc. Int. Symp. ASYNC*, 2001, pp.173–182.
14. M. Davies et al., "A 72-Port 10G ethernet switch/ router using quasi-delay-insensitive asynchronous design," in *Proc. Int. Symp. ASYNC*, 2014, pp.103–104.
15. P. Merolla et al., "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.
16. B. V. Benjamin et al., "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," in *Proc. IEEE*, vol. 102, no. 5, pp. 699–716, 2014.
17. S. B. Furber et al., "Overview of the SpiNNaker system architecture," *IEEE Trans. Comput.*, vol. 62, no. 12, pp. 2454–2467, 2013.

asynchronous designs can easily be specified with existing synchronous languages, including SystemC, SystemVerilog, and VHDL, to provide a seamless "on-ramp" for industrial designers. However, given the primacy of the clock and centralized control in these styles, typically extensions and enhancements are required. The latter approach aims to provide entirely new entry specification languages and/or synthesis flows, that are better tailored to an asynchronous paradigm.

Caltech Synthesis Method. One of the earliest and most influential asynchronous modeling and design flows comes from Alain Martin's group at Caltech from the 1980s [16]. The block-structured high-level entry language uses the notion of interacting processes, that communicate through blocking synchronization, augmented with constructs from Hoare's "communicating sequential processes" (CSP) language and Dijkstra's guarded commands. A synthesis method based on stepwise refinement has been proposed, using four-phase communication, where the processes are mapped to primitive abstract components whose handshaking protocols are then elaborated and scheduled. Several asynchronous RISC processors and microcontrollers have been modeled and synthesized using this approach (see "Processors and Architecture" sidebar). Alternative approaches have been proposed targeting two-phase communication, including a CSP-based approach by Ebergen based on trace theory and delay-insensitive decomposition [17], and an occam-based compilation approach by Brunvand and Sproull.

Philips' Tangram Compiler. In the late 1980s, at Philips Research (Eindhoven), the Tangram language was developed by Kees van Berkel [12]. While also block-structured and based on CSP, it uses an entirely different synthesis flow, modeled more on software compilers. In the first step, a syntax-directed translation directly maps processes to intermediate hardware components, e.g., small control cells and datapath blocks. In the second step, these components are individually mapped to gates, after applying some limited peephole optimizations. A Tangram-based tool flow was used extensively at Philips until the mid-2000s, with back-end physical design performed using commercial synchronous tools (Synopsys, Cadence, Magma), and bundled data timing constraints enforced through directives. This approach was the first fully-automated and industrial-quality asynchronous tool flow, with extensive support for testability, profiling and early power and performance estimation. A Philips-incubated startup company, Handshake Solutions, then migrated the language into a new variant, Haste. A public-domain version, Balsa [18], with some new enhancements, was used for the synthesis of a DMA controller for an asynchronous ARM processor, Amulet 3i (see "Processors and Architecture" sidebar), as well as for an ARM-compatible core, SPA, targeting smart card applications.

Pipelined System Synthesis and Optimization. An entirely different approach is directly to target pipelined microarchitectures. An initial pipelined system and its topology are specified using a set of basic components such as fork/join, split/merge, conditional operation, and loop control. Automated analysis and optimization are then applied to improve system-level performance through path rebalancing, including slack matching (i.e., FIFO insertion), splitting and coalescing of pipeline stages, loop unrolling and automatic pipelining [19], [20]. An automated tool flow, called Proteus [20], was developed at Fulcrum Microsystems for high-performance asynchronous ASICs, incorporating a number of these optimizations, as well as a high-level CSP-based HDL (CAST) and an RTL translator (CAST2RTL). The tool was recently migrated to Intel, and used to design its FM5000/6000 series Ethernet switch chips. An earlier influential approach, called *desynchronization* [21], provides a direct translation method from a synchronous pipelined netlist to an asynchronous implementation.

Null Convention Logic (NCL) Approaches. Several automated tool flows have been developed for NCL-based systems. A complete compiler which incorporates back-end synchronous tools was developed at Theseus Logic [8], using a standard synchronous-style HDL as an entry specification language. More recent NCL flows have made significant advances in automated CAD tool integration, support for semi-custom asynchronous libraries, and logic optimization and technology mapping [22], [23].

Other Complete Design Flows. An integrated asynchronous synthesis flow from the startup company, Tiempo, uses high-level Transaction Level Modeling (TLM), based on SystemVerilog, for specification entry [24]. The tool compiles to a gate-level netlist, which is then synthesized to layout using commercial synchronous tools.

Migration of Asynchrony into Synchronous Flows. Synchronous architectures and automated tool

Testing

Testing of asynchronous circuits provides both challenges and opportunities which are distinct from the testing of clocked circuits. There has been a body of recent research, for various fault models, test structures, pattern generation, and commercial application.

The absence of a global clock means that an asynchronous design cannot be slowed down simply by using a lower clock rate. In addition, many asynchronous datapaths use level-sensitive latches instead of edge-triggered flip-flops assumed by most synchronous test approaches. Moreover, asynchronous controllers, such as generated by burst-mode and Petri-net based techniques, typically store state on combinational feedback wires or using sequential C elements, rather than using flip-flops. An advantage of some asynchronous circuits, however, is that they exhibit the useful property of *self checking*, entering a deadlock state when subjected to certain stuck-at faults. As a result, fault diagnosis can take advantage of this additional failure mode.[1,2]

Given these distinguishing attributes, much effort within the asynchronous community over the past two decades has focused on developing new test techniques. Early work targeted scan testing for datapaths based on Sutherland's micropipelines.[3,4] The key idea was to modify pipeline latches and their controllers to introduce a clocked scan mode of operation. A similar full-scan approach was used commercially at Philips Semiconductors, where it was incorporated into their Tangram asynchronous design flow, and provided test quality equal to that of their commercial synchronous flow (i.e. over 99% stuck-at fault coverage).[5] Philips used an alternative asynchronous design-for-testability strategy, based on I_{DDQ} testing, to detect bridging faults.[6]

While full-scan approaches were promising for micropipelines, the overhead can be unacceptably high when fine-grained high-speed pipelines with shallow stages are used. A partial scan approach was developed for the commercial synthesis flow at Theseus Logic, using synchronous techniques, which yielded 100% stuck-at fault coverage.[7] A more recent approach by Shi et al.,[8] targeting Mousetrap pipelines, entirely eliminates the need for scan, relying instead on two interesting properties specific to many high-speed asynchronous pipelines. First, the datapath uses *normally-transparent* latches instead of flip-flops, thereby allowing an entire multi-stage pipeline to be placed in test mode as a single combinational flow-through logic block. Second, the asynchronous control circuits are *inherently self-checking* for a large fraction of stuck-at faults (see above), thereby leaving only a small number of faults to expose through test pattern application.

A novel approach to testing delay faults—in particular, timing constraint violations—in asynchronous pipelines has also been proposed.[9] Unlike synchronous approaches, very little testing hardware is added—in fact, none is required for linear pipelines—yet "at-speed" delay faults can be activated using only low-speed test equipment.

References

1. I. David, R. Ginosar, and M. Yoeli, "Self-timed is self-checking," *J. Electron. Testing*, vol. 6, no. 2, pp. 219–228, 1995.
2. S. J. Piestrak and T. Nanya, "Towards totally self-checking delay-insensitive systems," in *Proc. 25th Int. Symp. Fault-Tolerant Comput.*, 1995, pp. 228–237.
3. A. Khoche and E. Brunvand, "Testing micropipelines," in *Proc. Int. Symp. ASYNC*, 1994, pp. 239–246.
4. O. A. Petlin and S. B. Furber, "Built-in self-testing of micropipelines," in *Proc. Int. Symp. ASYNC*, 1997, pp. 22–29.
5. F. te Beest et al., "Automatic scan insertion and test generation for asynchronous circuits," in *Proc. ITC*, 2002, pp. 804–813.
6. M. Roncken, "Defect-oriented testability for asynchronous ICs," in *Proc. IEEE*, vol. 87, no. 2, pp. 363–375, 1999.
7. A. Kondratyev, L. Sorensen, and A. Streich, "Testing of asynchronous circuits by "inappropriate" means: Synchronous approach," in *Proc. Int. Symp. ASYNC*, 2002, pp. 171–180.
8. F. Shi et al., "Test generation for ultra-high-speed asynchronous pipelines," in *Proc. ITC*, 2005.
9. G. Gill et al., "Low-overhead testing of delay faults in high-speed asynchronous pipelines," in *Proc. Int. Symp. ASYNC*, 2006, pp. 46–56.

flows have also been developed which adopt asynchronous ideas of communicating back pressure and robust handling of long global wires into a clocked system, including latency-insensitive design [25] and synchronous elastic architectures [26].

Controller Synthesis Tools. The two asynchronous controller styles, burst-mode and Petrify, discussed earlier (see "Synthesis and Optimization") are each supported by automated public-domain tool flows. The MINIMALIST package, for burst-mode synthesis, includes algorithms for optimal state encoding and hazard-free two-level logic minimization, scripts to explore design-space tradeoffs, support for automatic insertion of initialization circuitry, and mapping to a Verilog netlist [2]. The Petrify tool [7] includes a wide range of algorithms for state encoding, logic optimization and technology mapping.

Testing. Support for testability is a critical component of any tool flow. Asynchronous design offers unique challenges, given the lack of a global clock and the requirement of hazard-free design. An overview of recent advances is highlighted in the "Testing" sidebar.

Performance and timing analysis

Performance and timing analysis are important components in developing optimized designs and practical tool flows. Asynchronous circuits and systems, with their lack of fixed-rate clock, and their use of fine-grained concurrency between small interacting components, pose distinct challenges and opportunities for such analysis.

Several general approaches have been proposed to analyze the operation of a concurrent system, which is modeled in the form of a Petri net or Event-Rule system. There are two main alternative approaches to model event timing: 1) using stochastic models (e.g., exponential delay), [27], [28] or 2) using min-max bounds [29]. The former approach provides metrics for average-case throughput, which can be used both to derive early performance estimates and to guide performance-driven optimization. The latter approach, which identifies min/max bounds on the separation of pairs of events over the entire operating lifetime of a system, provides useful timing information as well as formal event ordering guarantees that enable hardware simplification. Other approaches target the specialized problem of performance analysis of pipelined asynchronous systems [19], [20].

Formal verification and design validation

From a formal perspective, asynchronous circuits are instances of concurrent systems, and general techniques for this domain have been effectively used. The application of trace theory was pioneered by Dill [30], which models a concurrent system as an interaction of finite automata, and verifies basic safety properties through a conformance relation, with extensions to handle liveness and fairness properties using infinite automata. The focus is on speed-independent, QDI and DI circuits, and the method was used to verify arbiters, queues, and other systems. Other approaches use reachability analysis and model checking, including techniques to incorporate timing constraints, to validate implementations, such as are included in the Petrify tool [7], which can also formally check the consistency and well-formedness of specifications. Simulation, and functional and timing validation, techniques have also been developed for industrial flows at Tiempo [24] and Fulcrum/Intel [20].

Asynchronous design in the past decade or so has experienced a tremendous surge of interest, as designers grapple with the challenges of deep-submicron technology and large-scale heterogeneous system integration. Several significant inroads have been breached into the mainstream commercial world, both through major established companies and diverse startups. These have demonstrated benefits in important application areas, such as embedded microcontrollers, networks-on-chip, and high-performance Ethernet switch chips and FPGA's. There have also been promising advances in more radical emerging applications, such as ultra-low-energy design, neuromorphic computing, continuous-time DSP's, handling of extreme environments, nanomagnetics, flexible electronics, and energy harvesting. The inherent plug-and-play assembly, on-demand operation, and flexible communication strategies of the asynchronous regime fit well in a world where variability and unpredictability are first-class concerns, and where support for large-scale assembly of heterogeneous parallel architectures is becoming a critical requirement.

Design challenges still remain, however. The historical lack of commercial development of asynchronous CAD tools, coupled with some of the unique features of asynchronous systems, has led researchers to several alternative directions. At current standing, automated tool flows for synthesis, optimization and testability have been developed and used at several companies, but these custom in-house tools typically are specialized for particular design styles, and are not generally available to other researchers and designers. It is hoped, and anticipated, that this interest will drive the EDA industry to become partners in establishing asynchronous standards-based languages and widely-available commercial-grade tool flows. Inroads are also needed to better educate the microelectronics community, including the next generation of designers.

On occasion, designers have tried to harness asynchronous benefits by targeting an individual component for asynchronous implementation within a complex clocked system. However, the speed benefits of asynchrony, especially at the level of sub-clock-period, can sometimes be lost to the synchronization needed at mixed-timing interfaces, as happened with the RAPPID instruction length decoder. This has led many designers to believe that asynchronous design always has to be an all-or-nothing approach. However, several other examples—e.g., IBM's FIR filter, Intel/Fulcrum's Ethernet switch chips, CT-DSP's, GALS NoCs, etc.—have shown that mixing asynchronous with synchronous logic can, in many cases, be a highly-viable paradigm, and perhaps the best way forward for greater penetration into industry.

Finally, there is much at stake with emerging computing technologies, which represent both an opportunity and a challenge to asynchronous design. While the precise role of asynchronous design in technologies such as quantum cellular automata, nanomagnetics, self-assembled molecular electronics, etc., is as yet unclear, it is generally believed that some form of asynchrony will inevitably be required to enable these novel computing paradigms.

Acknowledgment

The authors appreciate the funding support of the National Science Foundation under Grants CCF-1219013, CCF-0964606, and OCI-1127361.

■ References

[1] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 14, no. 8, pp. 986–997, 1995.

[2] R. M. Fuhrer et al., *MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines*. Dept. Comp. Sci., Columbia Univ., Tech. Report CUCS-020-99. [Online]. Available: http://hdl.handle.net/10022/AC:P:29316; http://www.cs.columbia.edu/~nowick/asynctools

[3] D. S. Kung, "Hazard-non-increasing gate-level optimization algorithms," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD 92)*, 1992, pp. 631–634.

[4] S. M. Nowick and C. W. O'Donnell, "On the existence of hazard-free multi-level logic," in *Proc. 9th IEEE Int. Symp. Adv. Res. Asynch. Circuits Syst. (ASYNC 03)*, 2003, pp. 109–120.

[5] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," in *Proc. IEEE Int. Conf. Comput. Design (ICCD 91)*, 1991, pp. 192–197.

[6] A. Marshall, B. Coates, and P. Siegel, "Designing an asynchronous communications chip," *IEEE Design & Test*, vol. 11, no. 2, pp. 8–21, 1994.

[7] J. Cortadella et al., "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEEE Trans. Inf. Syst.,* vol. E80-D, no. 3, pp. 315–325, 1997. [Online]. Available: http://www.lsi.upc.edu/~jordicf/petrify

[8] M. Ligthart et al., "Asynchronous design using commercial HDL synthesis tools," in *Proc. 6th IEEE Int. Symp. Adv. Res. Asynch. Circuits Syst. (ASYNC 00)*, 2000, pp. 114–125.

[9] K. S Stevens, R. Ginosar, and S. Rotem, "Relative timing [asynchronous design]," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 1, pp. 129–140, 2003.

[10] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 1, no. 2, pp. 106–119, 1993.

[11] H. Jacobson et al., "High-level asynchronous system design using the ACK framework," in *Proc. 6th IEEE Int. Symp. Adv. Res. Asynch. Circuits Syst. (ASYNC 00)*, 2000, pp. 93–103.

[12] K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming.* Cambridge, U.K.: Cambridge University Press, 1993.

[13] J. Hansen and M. Singh, "A fast branch-and-bound approach to high-level synthesis of asynchronous systems," in *Proc. Int. Symp. Asynch. Circuits Syst. (ASYNC-10)*, 2010, pp. 107–116.

[14] R. M. Badia and J. Cortadella, "High-level synthesis of asynchronous systems: scheduling and process synchronization," in *Proc. European Design Automation Conf.*, 1993, pp. 70–74.

[15] J. Hansen and M. Singh, "Multi-token resource sharing for pipelined asynchronous systems," in *Proc. ACM/IEEE Design, Automation Test Europe Conf. (DATE 12)*, 2012, pp. 1191–1196.

[16] A. J. Martin, *Programming in VLSI: from Communicating Processes to Delay-Insensitive Circuits*, Dept. of Comp. Sci., California Inst. of Tech., Tech. Rep. CS-TR-89-1.

[17] J. C. Ebergen, "A formal approach to designing delay-insensitive circuits," *Distrib. Comput.*, vol. 5, no. 3, pp. 107–119, 1991.

[18] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer J.,* vol. 45, no. 1, pp. 12–18, 2002.

[19] G. Gill and M. Singh, "Automated microarchitectural exploration for achieving throughput targets in pipelined asynchronous systems," in *Proc. IEEE Symp. Asynch. Circuits Syst. (ASYNC 10)*, 2010, pp. 117–127.

[20] P. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for GHz asynchronous designs," *IEEE Design & Test*, vol. 28, no. 5, pp. 38–51, 2011.

[21] J. Cortadella et al., "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 10, pp. 1904–1921, 2006.

[22] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle—An RTL approach to asynchronous design," in *Proc. 18th IEEE Int. Symp. Asynch. Circuits Syst. (ASYNC 12)*, 2012, pp. 65–72.

[23] M. Moreira et al., "Semi-custom NCL design with commercial EDA frameworks: Is it possible?," in *Proc. 20th IEEE Int. Symp. Asynch. Circuits Syst. (ASYNC 14)*, 2014, pp. 53–60.

[24] A. Yakovlev, P. Vivet, and M. Renaudin, "Advances in asynchronous logic: From principles to GALS & NoC, recent industry applications, commercial CAD tools," in *Proc. ACM/IEEE Design, Automation and Test in Europe Conf. (DATE 13)*, 2013, pp. 1715–1724.

[25] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, 2001.

[26] J. Carmona et al., "Elastic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1437–1455, 2009.

[27] P. A. Beerel and A. Xie, "Performance analysis of asynchronous circuits using Markov chains," in *Proc. Concurrency Hardware Design*, 2002, pp. 313–344.

[28] P. B. McGee and S. M. Nowick, "Efficient performance analysis of asynchronous systems based on periodicity," in *Proc. CODES-ISSS*, 2005, pp. 225–230. [Online]. Available: http://www.cs.columbia.edu/~nowick/asynctools

[29] H. Hulgaard et al., "An algorithm for exact bounds on time separation of events in concurrent systems," *IEEE Trans. Comput.*, vol. 44, no. 11, pp. 1306–1317, 1995.

[30] D. L. Dill, *Trace Theory for the Automatic Hierarchical Verification of Speed-Independent Circuits*. Cambridge, MA, USA: MIT Press, 1989.

Steven M. Nowick is a professor of computer science at Columbia University, New York, NY, USA. His research interests include the design and optimization of asynchronous and mixed-timing (i.e., GALS) digital systems, scalable and low-latency on-chip interconnection networks for shared-memory parallel processors and embedded systems, extreme low-energy digital systems, neuromorphic computing, and variation-tolerant global communication. He has a PhD in computer science from Stanford University, Stanford, CA, USA. He is a Fellow of the IEEE.

Montek Singh is an associate professor of computer science at the University of North Carolina at Chapel Hill, NC, USA. His research interests include asynchronous and mixed-timing circuits and systems; CAD tools for design, analysis, and optimization; high-speed and low-power VLSI design; and applications to emerging computing technologies, energy-efficient graphics, and image sensing hardware. He has a PhD in computer science from Columbia University, New York, NY, USA.

■ Direct questions and comments about this article to Steven M. Nowick, Department of Computer Science, Columbia University, New York, NY 10027 USA; nowick@cs.columbia.edu; or to Montek Singh, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599 USA; montek@cs.unc.edu.