A Survey and Taxonomy of GALS Design Styles

Paul Teehan, Mark Greenstreet, and Guy Lemieux
University of British Columbia

Editor's note
The authors categorize GALS design styles into three distinct classes: pausible clocks, asynchronous interfaces, and loosely synchronous interfaces. They present examples, and discuss advantages and relative pitfalls for each design style. Engineers interested in GALS-style integration of synchronous IP blocks and cross-domain communications may find the concepts and taxonomy presented here very useful.
—Sandeep Shukla, Virginia Tech

SINGLE-CLOCKED DIGITAL SYSTEMS are largely a thing of the past. Although most digital circuits remain synchronous, many designs feature multiple clock domains, often running at different frequencies. Using an asynchronous interconnect decouples the timing issues for the separate blocks. Systems employing such schemes are called globally asynchronous, locally synchronous (GALS). Figure 1 shows an example. GALS designs offer increased ease of functional-block reuse, simplified timing closure, and power advantages due to heterogeneous clocking.

To minimize time to market, large SoC designs must integrate many functional blocks with minimal design effort.[1] These blocks are usually designed using standard synchronous methods and often have different clocking requirements. A GALS approach can facilitate fast block reuse by providing wrapper circuits to handle interblock communication across clock domain boundaries. SoCs may also achieve power savings by clocking different blocks at their minimum speeds. For example, Scott et al. describe the advantages of GALS design for an embedded-processor peripheral bus.[2]

High-performance microprocessors face similar pressures. As transistor counts and clock frequencies increase, distributing a low-skew global clock becomes increasingly more difficult. Iyer and Marculescu studied GALS-based microprocessors and concluded that they could gain power advantages by allowing fine tuning of the supply voltages and clock speeds for different functional blocks and by eliminating the need for a global, low-skew clock.[3] Semeraro et al.;[4] Zhu, Albonesi, and Buyuktosunoglu;[5] Chattopadhyay and Zilic;[6] and others have further studied dynamic voltage and frequency scaling using a GALS approach.

Crossing clock domains is the central problem in GALS designs. If the data for a flip-flop or latch comes from another timing domain, it could potentially violate the setup and hold requirements. Such a timing violation could cause a metastable output, in which the

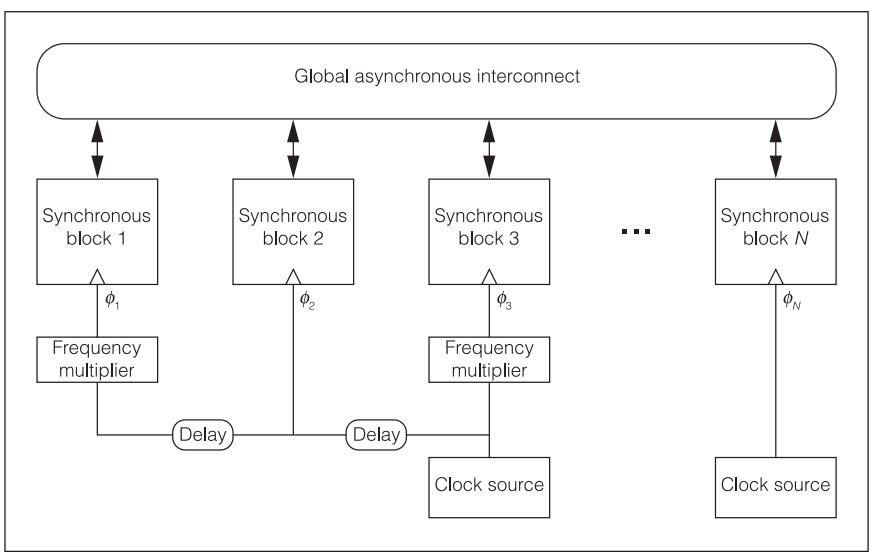

Figure 1. High-level diagram of a globally asynchronous, locally synchronous (GALS) system.

0740-7475/07/$25.00 © 2007 IEEE Copublished by the IEEE CS and the IEEE CASS

voltage level may be indeterminate for an unbounded length of time before settling to a valid level.[7] However, it's possible to minimize the probability of metastability failures by using synchronizer circuits, which can be as simple as one or more flip-flops connected in series. Figure 2 shows a common two-flop synchronizer. Failure probability drops exponentially with settling time or, equivalently, with the number of flip-flops in the chain. Thus, synchronizers can provide mean times between failures (MTBFs) of millions of years or more if properly designed.[8]

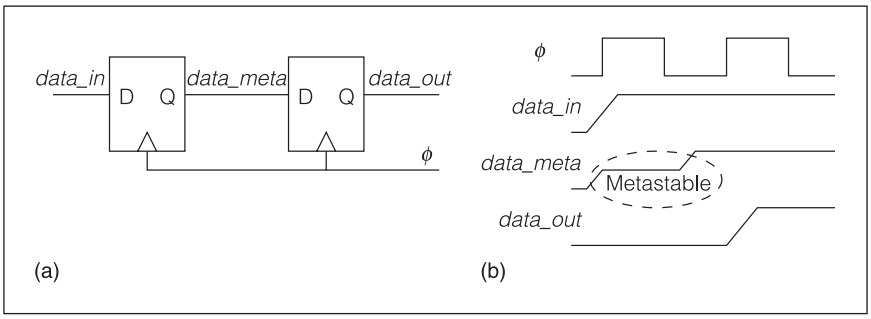

Figure 2. A two-flop synchronizer, showing metastability: circuit (a) and timing diagram (b).

We classify GALS design styles according to the methods they use to transfer data between timing domains. In this article, we describe some design examples and introduce our taxonomy of these techniques.

Taxonomy and design examples

We identify three broad categories of GALS design styles, as Figure 3 shows: pausible clock, asynchronous, and loosely synchronous.

The *pausible-clock* design style relies on locally generated clocks that can be stretched or paused either to prevent metastability or to let a transmitter or receiver stall because of a full or empty channel. A ring oscillator typically generates the clocks. The Integrated Systems Laboratory at ETHZ (Swiss Federal Institute of Technology Zurich) has implemented several chips featuring pausible clocks,[9] including a cryptography chip.[10] Special wrapper circuits interface between synchronous blocks, such that each wrapper includes a pausible-clock generator.

The *asynchronous* design style involves the general case in which no timing relationship between the synchronous clocks is assumed. Such designs are maximally flexible with respect to timing. For example, Fulcrum Microsystems' Nexus architecture includes an asynchronous crossbar switch that handles communication between blocks operating at arbitrary clock frequencies.[11]

The *loosely synchronous* design style is for cases in which there is a well-defined, dependable relationship between clocks. It's possible to exploit the stability of these clocks to achieve high efficiency while simultaneously providing tolerance for the large amounts of skew inherent in global interconnects. Messerschmitt[12] has proposed a taxonomy of commonly occurring timing relationships:

- *Mesochronous.* The sender and receiver operate at exactly the same frequency with an unknown yet stable phase difference. Intel's 80-core processor employs a mesochronous design.[13] It uses synchronous tiles and a skew-tolerant network-on-chip (NoC) interconnect scheme driven by one stable global clock.
- *Plesiochronous.* The sender and receiver operate at the same nominal frequency but may have a slight frequency mismatch, such as a few parts per million, which leads to drifting phase. Gigabit Ethernet is a common example.
- *Heterochronous.* The sender and receiver operate at nominally different clock frequencies.

An interesting subset of heterochronous relationships is the case of rationally related clock frequencies in which the receiver's clock frequency is an exact rational multiple of the sender's, and both are derived from the same source clock such that there is a predictable periodic phase relationship. We refer

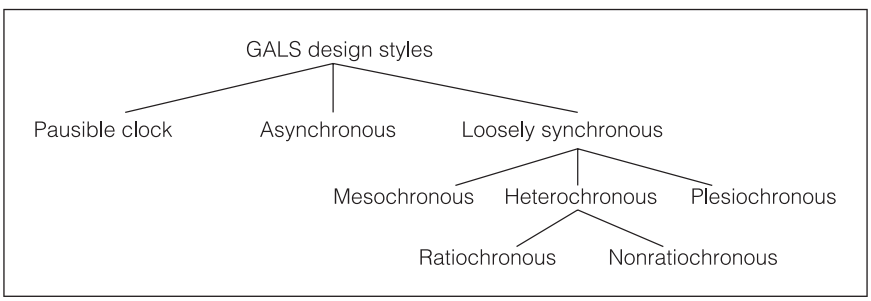

Figure 3. Taxonomy of GALS design styles.

to this relationship as *ratiochronous*, a term which we believe was first used by Heath, Burleson, and Harris.[14]

For each of these three GALS design styles, we describe a simplified example that provides one-way communication between transmitter and receiver blocks. The blocks operate synchronously using two different clocks and are connected together using a FIFO buffer that is robust and free of metastability. (For other examples of this use of FIFO buffers, see the works of Sparsø and Sutherland.[15,16]) This FIFO buffer can have almost any capacity, including just one data item, but this may affect throughput. To send a data item, the transmitter asserts *put* and drives *data_in*. The FIFO buffer accepts the data on the rising edge of *put* and lowers *ok_to_put*. If this operation fills the FIFO buffer, *ok_to_put* remains low until some data is removed. On the receiver side, the FIFO buffer asserts *ok_to_take* when data is available. To remove a data item, the receiver latches *data_out* and asserts *take*. The FIFO buffer lowers *ok_to_take* until new data is available. If the FIFO buffer is empty, *ok_to_take* remains low until new data is inserted.

Pausible clocks

The first use of the term *GALS* was by Chapiro in his 1984 doctoral dissertation.[17] He proposed using pausible clocks to enable separate clock domains to communicate without metastability. With Chapiro's approach, each locally synchronous block generates its own clock with a ring oscillator. Each ring oscillator's period is set according to the speed requirements of the block it drives.

Two potential advantages of pausible clocking are robustness and power. Pausing delays a clock's sampling edge until after the arrival of data from the other domains, thus avoiding metastability altogether. Also, pausing the clock of a block awaiting communication prevents that block from dissipating dynamic power. Presumably, V_{DD} can be lowered during prolonged stalls to reduce static power as well. Hence, this style may be useful in power-critical designs.

Example

Figure 4 shows an example of pausible clocks. Each ring oscillator contains a NAND gate to control clock pausing. The transmitter clock should be allowed to run if it is currently high, if the FIFO buffer can accept a new value (*ok_to_put* asserted), or if the transmitter is not attempting to send (*ready_to_put* is low). Likewise, the receiver clock should be allowed to run if it is currently high, if the FIFO buffer has data ready (*ok_to_take* asserted), or if the receiver is not attempting to read new data (*ready_to_take* is low). In this manner, a rising clock edge acknowledges that it is OK to proceed.

The timing diagram in Figure 4 shows the transfer of two consecutive data items. Assume the FIFO buffer is initially empty and can hold only 1 datum. The receiver is ready (*ready_to_take* asserted), but its clock is paused because the FIFO buffer is empty. The transmitter is ready, having driven *tx_data* and *ready_to_put* at the end of the last cycle. While the transmitter clock is low, latch L_T is transparent, but the AND gate keeps *put* low. When the FIFO buffer is ready (*ok_to_put* asserted), a rising transmitter clock edge is produced, which asserts *put*, fills the FIFO buffer with the first datum, and lowers *ok_to_put*. At this point, the transmitter clock pauses because it is immediately ready with a second datum (*ready_to_put* asserted) and the FIFO buffer is full. Meanwhile, the assertion of *ok_to_take* restarts the receiver's clock. The receiver latches *rx_data* and then asserts *take*, signaling the FIFO buffer that the data has been removed. Because the FIFO buffer is no longer full, *ok_to_put* goes high, which restarts the transmitter clock so that the second datum can be transmitted.

Extensions

The simple case of a single transmitter and receiver can be generalized to designs in which each block communicates with multiple other blocks. Yun and Donohue[18] and Yun and Dooply[19] developed such a system using ring- and bus-based arbiters to select an input to service, and using a mutual-exclusion (mutex) element to gate the clock. They designed these circuits to pause the clock until metastability resolves to a stable, logical value (that is, high or low). Bormann and Cheung developed similar designs that avoid the use of arbiters and polling by explicitly scheduling transfers.[20]

Clock tree latency must be considered in GALS designs. If the latency to distribute a clock is larger than a single clock cycle, invalid operations may occur after the clock was supposed to have stopped. Mekie, Chakraborty, and Sharma propose adding artificial delays between the GALS interface and the synchronous block to account for clock tree delays.[21] Mullins and Moore present an excellent examination of clock distribution and other challenges for pausible-clock designs.[22]

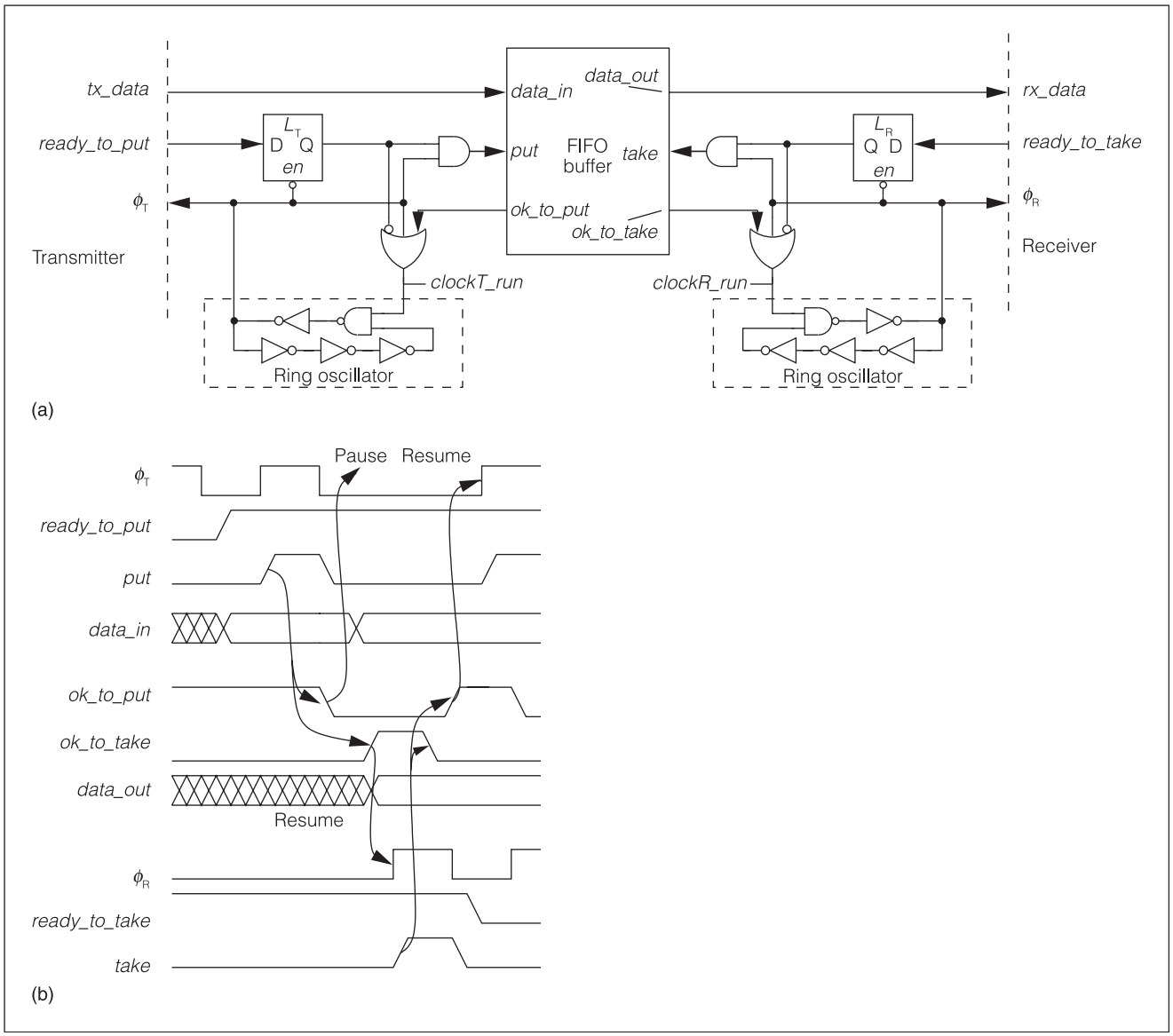

Figure 4. Pausible-clock GALS design style: circuit (a) and timing diagram (b).

Design issues

Pausible clocking encapsulates crucial timing constraints in the clock generator wrappers, simplifying design reuse. By controlling the receiver's clock, these interfaces ensure that data arriving at the receiver satisfies the receiver's timing requirements, thus completely avoiding metastability. Once this interface wrapper IP has been verified, it can be reused for many different local blocks without the need for further timing analysis.

Gurkaynak et al. noted that designing ring oscillators for robustness and good performance was a major difficulty in their GALS research.[9] They concluded that pausible clocking "remains a niche technology at best."[9] For example, the clock period can have high jitter, varying significantly from cycle to cycle as it restarts from a pause.[23] This jitter can be amplified by the clock distribution network, further cutting into the timing margin.

A potential advantage of ring oscillator clocks is that variations in the clock period should track variations in logic-gate delays across a range of operating conditions. Unfortunately, standard CAD tools do not account for this behavior during analysis, and they might force conservative, worst-case designs.

Asynchronous interfaces

The second GALS design style is the asynchronous interface. This method uses circuits known as synchronizers to transfer signals arriving from an outside timing domain to the local timing domain. Although simple asynchronous interfaces suffer from low throughput, this limitation can be overcome with careful designs.

Example

Figure 5 shows an asynchronous GALS design example. The timing diagram shows the transfer of two data values from the transmitter to the receiver, assuming an initially empty FIFO buffer. In this circuit, the FIFO buffer handshake signals, *ok_to_put* and *ok_to_take*, may be asserted at any time relative to the transmitter or receiver clocks, respectively. This design uses two flip-flops to synchronize a signal with the local clock and avoid metastability. To account for the synchronizer's delay, the *put_wait* signal prevents the transmitter from sending until the FIFO buffer status following the previous *put* has propagated through the synchronizer. The *take_wait* signal serves the same function for the receiver. This simplistic design can transfer at most one datum for every three cycles of transmitter clock ϕ_T or receiver clock ϕ_R, whichever is slower.

Extensions

Seizovic increased the throughput of an asynchronous interface by pipelining the synchronization operations through a FIFO buffer along with the data.[24] The probability of synchronization failure is determined by the total time the data is in the FIFO buffer, allowing very low failure probabilities with high data throughput rates. This design allows a throughput of one data item for every cycle of clock ϕ_T or clock ϕ_R, whichever is slower. Boden et al. used Seizovic's pipeline synchronizers in the design of the Myrinet high-speed network hardware.[25]

More recently, Chelcea and Nowick proposed a general family of low-latency synchronizing FIFO buffers.[26] The key idea of their design is to detect when the FIFO buffer is nearly empty—that is, contains fewer values than the number of flops in the synchronizer—or nearly full. The signals for these conditions are synchronized along with the usual empty and full signals. As long as the synchronized version of nearly empty is false, the receiver may take a value every cycle. Otherwise, it can revert to using the *empty* signal to remove the last items from the FIFO buffer at a slower rate. A similar argument applies to the transmitter. This lets the FIFO buffer transfer data at the full rate of the transmitter or receiver, whichever is slower. This design supports arbitrary combinations of synchronous and asynchronous communicating blocks as well as long interconnect delays, making it well-suited for large SoC designs with many different and perhaps time-varying clock frequencies.

Several recent designs attempt to smoothly integrate synchronous designs into an asynchronous network with minimal design effort. For example, the PivotPoint design uses delay-insensitive codes to transmit values between local blocks and the PivotPoint crossbar.[27] Similar approaches are described elsewhere.[28,29]

Design issues

Asynchronous interfaces offer the most flexibility and probably the easiest integration into existing CAD flows. The main concern is the modeling and validation of the synchronizer circuits and the impact of their delay. As described by Kinniment, Heron, and Russell,[30] real synchronizers have more complicated behavior than predicted by simple textbook models, and circuit simulators such as Spice do not have the numerical accuracy to verify acceptable reliabilities. Recently developed simulation methods address this problem.[31,32] We expect that mainstream GALS designs will use synchronizers that are encapsulated in IP blocks such as those provided by Fulcrum Microsystems[27] or Silistix,[2] with the synchronizers in these blocks validated by the vendors using techniques such as those presented by Yang and Greenstreet.[31,32]

A rule of thumb for synchronizer design is that at least 40 gate delays should be budgeted for metastability to resolve to a stable, logical value.[33] For a 0.13-micron process with a 60-ps gate delay, synchronization adds about 2.5 ns of delay when crossing timing domains. Thus, we expect the asynchronous GALS style to find widespread use in SoC designs that can tolerate the extra latency of synchronization or that have low clock frequencies (that is, few cycles of synchronization latency). Higher-performance designs will require the loosely synchronous styles described next.

Loosely synchronous interfaces

The third GALS design style, loosely synchronous interfaces, arises when some bounds on the frequen-

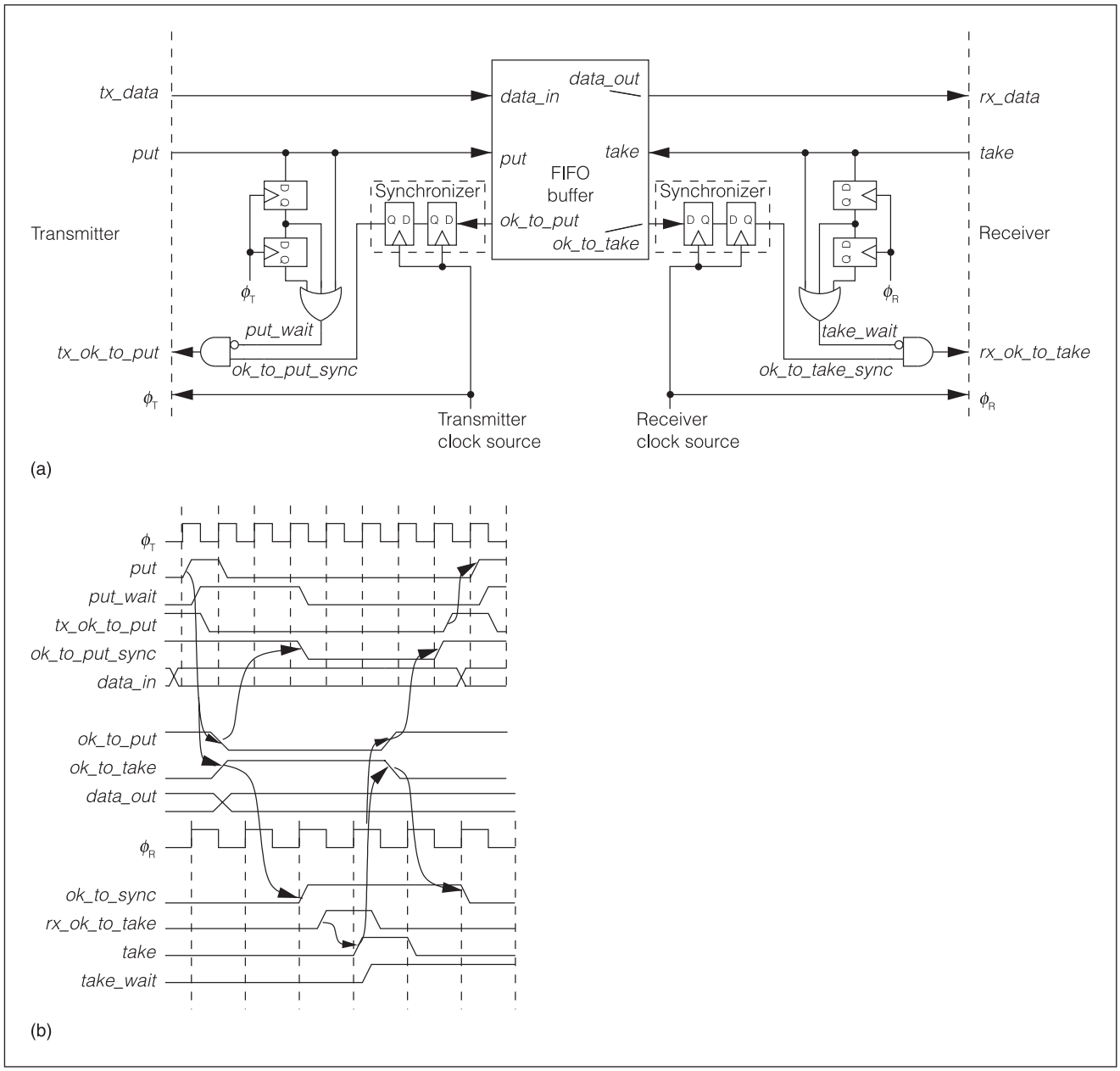

Figure 5. Asynchronous GALS design style employing synchronizers: circuit (a) and timing diagram (b).

cies of communicating blocks are known. In this style, the designer exploits these bounds to ensure that timing requirements are met. This style requires timing analysis on the paths between the sender and receiver and is less amenable to dynamic changes in the clock frequency. However, this analysis makes handshaking unnecessary during data transfer, so the resulting circuits can achieve higher performance and have more deterministic latencies than those of the other methods.

Example

A loosely synchronous design exploits one of the known timing relationships we described earlier. The simplest case is a mesochronous relationship, in which the frequencies are exactly matched and there is a stable but unknown phase difference. This commonly occurs when the clocks are derived from the same source but the latency of delivery to each block differs.

The mesochronous example shown in Figure 6 is based on the Stari (Self-Timed at Receiver's Input)

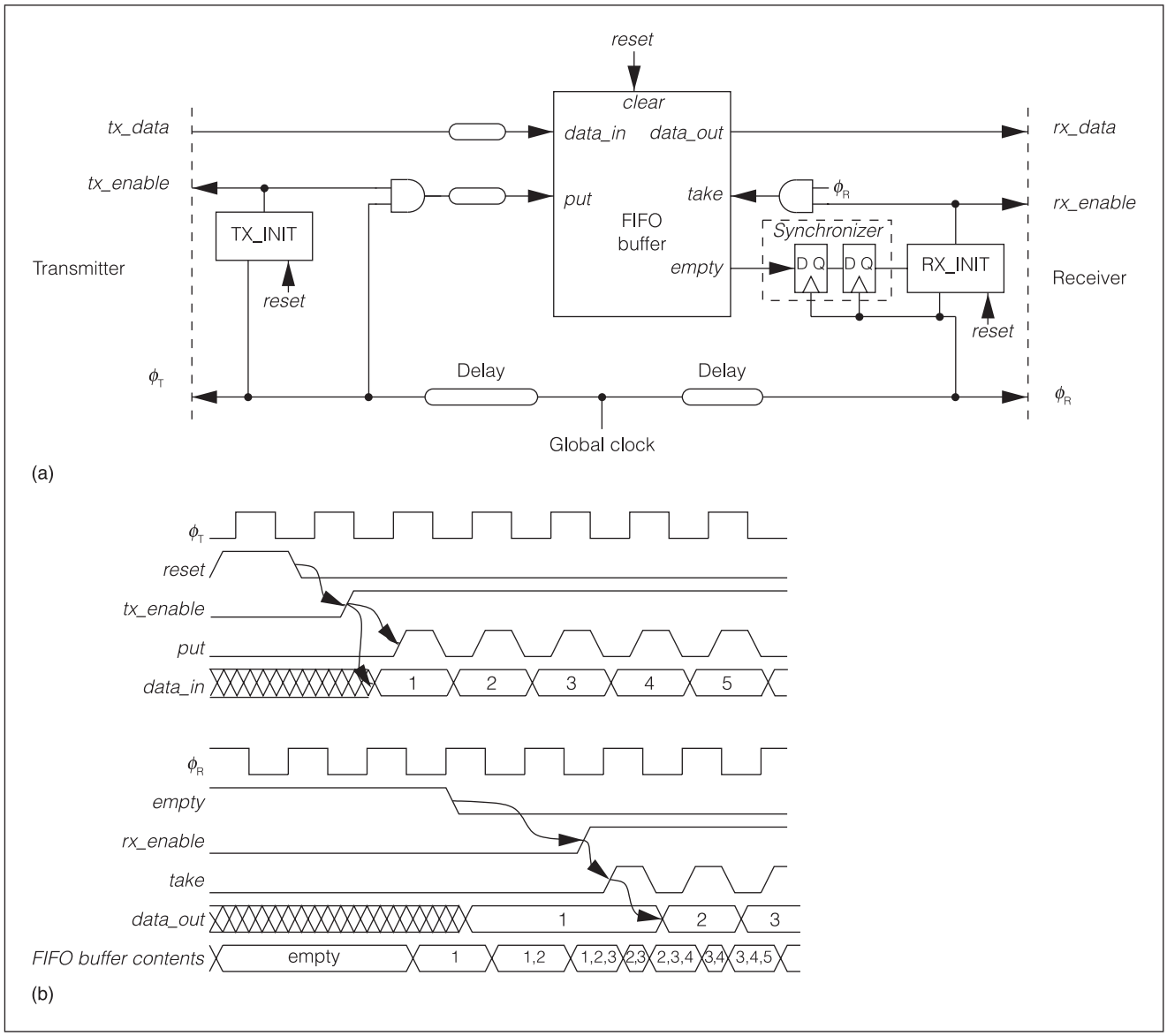

Figure 6. Loosely synchronous, mesochronous GALS design style: circuit (a) and timing diagram (b).

scheme,[34] in which clocks ϕ_T and ϕ_R are derived from the same source. The receiver uses a self-timed FIFO buffer to compensate for the phase difference. The key to high-performance operation is to initialize the FIFO buffer to be half full. During operation, the transmitter puts one datum into the FIFO buffer every cycle, and the receiver takes one datum. Neither needs to check the FIFO buffer status signals (the FIFO buffer is assumed to be fast enough), but the FIFO buffer will remain within ± 1 data item of half full because the frequencies are matched. If needed, higher-level flow control information can be embedded in the data (for example, by defining a valid bit) rather than stopping the transmitter.

To get the FIFO buffer half full, special initialization is needed. Initially, a global *reset* signal is asserted, which may need to be synchronized. The TX_INIT block awaits a fixed number of cycles until the reset is guaranteed to have completed everywhere, and then enables the transmitter by asserting *tx_enable*. The transmitter begins sending data. After the first reset data item arrives, *empty* goes low. Because the transmitter and receiver can have arbitrary skew, this change of *empty* is asynchronous with respect to the receiver's clock and must be synchronized. After the synchronizer latency, the RX_INIT block receives the signal, awaits any additional cycles necessary for the FIFO buffer to reach the half-full state, and asserts

rx_enable. On the next receiver cycle, the receiver begins removing data items at the same rate that the transmitter sends them, and no further synchronization is required.

Extensions

If the timing variations between the sender and receiver are relatively small, the mesochronous interface can be highly optimized. Chakraborty and Greenstreet provide a similar interface with a single-stage, clocked FIFO buffer, which can tolerate nearly two clock periods of phase uncertainty between the sender and receiver.[35] The FIFO buffer clock comes from an event-driven circuit that watches the transmitter and receiver clocks and generates a clock pulse during a safe timing window.

In ratiochronous designs, the synchronous blocks use clocks that are exact rational multiples of one another. Mesochronous methods can be extended to handle this case. For example, a design could include blocks that operate at 300 MHz and 700 MHz, both derived from multiples of a 100-MHz reference. In this case, the phase relationship between the two clocks varies in a predictable, periodic fashion. Chakraborty and Greenstreet presented a design that uses binary-rate multipliers for the faster of the transmitter or receiver to generate an approximation of the other clock,[35] which is input to the event-driven clock generator just mentioned.

An alternative approach is to allow normal transmission except when the data would arrive nearly coincident with the receiver's clock. Mekie et al. proposed preventing transmission on unsafe cycles by examining and modifying the communication protocol, exploiting the periodic relationship of the clock phases.[36] If the transmitter never transmits in an unsafe cycle, then a synchronizing interface is unnecessary. However, this solution depends on controlling the global skew between communicating blocks and leads to very stringent timing constraints.

In plesiochronous designs, the transmitter and receiver have clocks of closely matched frequencies. The phase differences between these clocks may slowly drift, leading to violations of the receiver's timing constraints. However, it is possible to detect when an unsafe state is approaching and take corrective action to move back to a safe state. Moreover, because the phase drift is slow, such events will be infrequent, and the speed of the corrective action is not critical. No synchronization is needed in a safe state, so there need not be a latency penalty during normal data transfers. (Implementations are presented elsewhere.[35,37])

If the transmitter and receiver are operating at unrelated but stable frequencies, then they can estimate each other's clock frequency. This estimate provides a rational multiple, enabling the use of ratiochronous methods. Then, plesiochronous techniques can handle the residual frequency mismatch. Chakraborty and Greenstreet present details and implementations of these approaches.[35]

Design issues

The need for high clock frequencies and low latency in high-performance designs will make them candidates for loosely synchronous techniques. However, to determine the optimal size of the FIFO buffers, timing analysis is necessary to bound how far the relative phase difference between the sender and receiver may drift. Although this type of timing analysis is not yet common for on-chip timing, it is standard when using interchip, source-synchronous communication (for example, synchronous DRAMs). This is an area where we expect CAD support to emerge as designers undertake chips with many timing domains.

GALS DESIGN STYLES BUILD on the extensive infrastructure of synchronous design while avoiding the problems of distributing a global, low-skew clock. A GALS methodology is a natural approach for SoC design, allowing the integration of independently designed blocks operating at different frequencies. Furthermore, some GALS approaches work easily with dynamic voltage scaling and other power reduction techniques.

Although pausible clocks are appealing for their elimination by construction of metastability failures, they do not fit well with existing CAD flows and do not scale well for designs with high-speed clocks. Pausible clocks are therefore unlikely to find widespread acceptance, although their ability to completely shut down during idle periods may make them attractive for low-power designs.

Fully asynchronous interfaces offer the greatest flexibility. Although some new CAD tool capabilities will be needed to support asynchronous interconnects, commercial tools are already evolving in this direction, with tools that check circuits spanning multiple clock domains for structural and protocol errors.[38] We expect further CAD and IP vendor support

to emerge as designers demand it for large SoC and NoC designs.

An additional problem with GALS designs that rely on arbiters or synchronizers is that they are inherently nondeterministic, which complicates design validation and test. To address these problems, some researchers have sought to make the timing of GALS designs deterministic.[14] Validation and test of GALS designs remains an important area for further research.

Mesochronous and other loosely synchronous techniques offer the highest performance by removing synchronization delays from latency-critical paths. However, these methods require timing analyses that standard CAD flows do not support. Thus, we expect that loosely synchronous styles will be used in performance-critical applications that justify the extra design effort. Furthermore, IP vendors can help ASIC designers to exploit loosely synchronous circuits by encapsulating them in predesigned interface blocks and by providing dedicated validation tools built atop standard timing-analysis software and other CAD tools.

GALS design faces a ''chicken-and-egg'' problem: most designers are unwilling to migrate until the CAD tools are available, and CAD companies are reluctant to provide the tools until the technology is widely used. However, the incentives for partitioning a design into smaller timing domains make some kind of GALS approach inevitable. Designs with fully asynchronous interfaces seem to require the least change to the local blocks while avoiding the need for new global timing-analysis tools. Accordingly, this style is likely to be the first to have adequate CAD support and thus become dominant. Pausible-clocking and loosely synchronous designs offer advantages for designers who need extremely low power or the highest possible performance. Historically, these designers have devised their own special-purpose tools, so we expect that they will likewise incorporate the GALS design styles that are most suitable to their needs. ■

Acknowledgments

We thank Brad Quinton for his perspective on CAD issues, and we thank the reviewers for their constructive comments.

■ References

1. R. Saleh et al., "System-on-Chip: Reuse and Integration," *Proc. IEEE*, vol. 94, no. 6, June 2006, pp. 1050-1069.
2. A.M. Scott et al., "Asynchronous On-Chip Communication: Explorations on the Intel PXA27x Processor Peripheral Bus," *Proc. 13th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 07), IEEE CS Press, 2007, pp. 60-72.
3. A. Iyer and D. Marculescu, "Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 158-168.
4. G. Semeraro et al., "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," *Proc. 8th Int'l Symp. High-Performance Computer Architecture* (HPCA 02), IEEE CS Press, 2004, pp. 29-40.
5. Y. Zhu, D.H. Albonesi, and A. Buyuktosunoglu, "A High Performance, Energy Efficient GALS Processor Microarchitecture with Reduced Implementation Complexity," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 05), IEEE CS Press, 2005, pp. 42-53.
6. A. Chattopadhyay and Z. Zilic, "GALDS: A Complete Framework for Designing Multiclock ASICs and SoCs," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 6, June 2005, pp. 641-654.
7. T.J. Chaney and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Trans. Computers*, vol. C-22, no. 4, Apr. 1973, pp. 421-422.
8. R. Ginosar, "Fourteen Ways to Fool Your Synchronizer," *Proc. 9th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 03), IEEE CS Press, 2003, pp. 89-96.
9. F.K. Gurkaynak et al., "GALS at ETH Zurich: Success or Failure?" *Proc. 12th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 06), IEEE CS Press, 2006, pp. 150-159.
10. J. Muttersbach, T. Villiger, and W. Fichtner, "Practical Design of Globally-Asynchronous Locally-Synchronous Systems," *Proc. 6th Int'l Symp. Advanced Research in Asynchronous Circuits and Systems* (ASYNC 00), IEEE CS Press, 2000, pp. 52-59.
11. A. Lines, "Asynchronous Interconnect for Synchronous SoC Design," *IEEE Micro*, vol. 24, no. 1, Jan.-Feb. 2004, pp. 32-41.
12. D.G. Messerschmitt, "Synchronization in Digital System Design," *IEEE J. Selected Areas in Communications*, vol. 8, no. 8, Oct. 1990, pp. 1404-1419.
13. S. Vangal et al., "An 80-Tile 1.28TFLOPS Network-on-Chip in 65 nm CMOS," *Proc. IEEE Int'l Solid-State Circuits Conf.* (ISSCC 07), IEEE Press, 2007, pp. 98-99, 589.

14. M.W. Heath, W.P. Burleson, and I.G. Harris, "Synchrotokens: A Deterministic GALS Methodology for Chip-Level Debug and Test," *IEEE Trans. Computers*, vol. 54, no. 12, Dec. 2005, pp. 1532-1546.
15. J. Sparsø, "Asynchronous Circuit Design – A Tutorial," *Principles of Asynchronous Circuit Design: A Systems Perspective*, J. Sparsø and S. Furber, eds., Kluwer Academic Publishers, 2001, pp. 1-152.
16. I.E. Sutherland, "Micropipelines," *Comm. ACM*, vol. 32, no. 6, June 1989, pp. 720-738.
17. D.M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems," doctoral dissertation, Dept. of Computer Science, Stanford Univ., 1984.
18. K.Y. Yun and R.P. Donohue, "Pausible Clocking: A First Step toward Heterogeneous Systems," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 96), IEEE CS Press, 1996, pp. 118-123.
19. K.Y. Yun and A.E. Dooply, "Pausible Clocking-Based Heterogeneous Systems," *IEEE Trans. Very Large Scale Integration (VSLI) Systems*, vol. 7, no. 4, Dec. 1999, pp. 482-488.
20. D.S. Bormann and P.Y.K. Cheung, "Asynchronous Wrapper for Heterogeneous Systems," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 97), IEEE CS Press, 1997, pp. 307-314.
21. J. Mekie, S. Chakraborty, and D.K. Sharma, "Evaluation of Pausible Clocking for Interfacing High Speed IP Cores in GALS Framework," *Proc. 17th Int'l Conf. VLSI Design*, IEEE CS Press, 2004, pp. 559-564.
22. R. Mullins and S. Moore, "Demystifying Data-Driven and Pausible Clocking Schemes," *Proc. 13th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 07), IEEE CS Press, 2007, pp. 175-185.
23. A.J. Winstanley, A. Garivier, and M.R. Greenstreet, "An Event Spacing Experiment," *Proc. 8th Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 02), IEEE CS Press, 2002, pp. 47-56.
24. J.N. Seizovic, "Pipeline Synchronization," *Proc. Int'l Symp. Advanced Research in Asynchronous Circuits and Systems* (ASYNC 94), IEEE CS Press, 1994, pp. 87-96.
25. N.J. Boden et al., "Myrinet: A Gigabit-Per-Second Local Area Network," *IEEE Micro*, vol. 15, no. 1, Jan.-Feb. 1995, pp. 29-36.
26. T. Chelcea and S.M. Nowick, "Robust Interfaces for Mixed-Timing Systems," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 8, Aug. 2004, pp. 857-873.
27. U. Cummings, "PivotPoint: Clockless Crossbar Switch for High-Performance Embedded Systems," *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 48-59.
28. B.R. Quinton, M.R. Greenstreet, and S.J.E. Wilton, "Asynchronous IC Interconnect Network Design and Implementation Using a Standard ASIC Flow," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 05), IEEE CS Press, 2005, pp. 267-274.
29. J. Bainbridge and S. Furber, "Chain: A Delay-Insensitive Chip Area Interconnect," *IEEE Micro*, vol. 22, no. 5, Sept./Oct. 2002, pp. 16-23.
30. D. Kinniment, K. Heron, and G. Russell, "Measuring Deep Metastability," *Proc. 12th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 06), IEEE CS Press, 2006, pp. 2-11.
31. S. Yang and M.R. Greenstreet, "Computing Synchronizer Failure Probabilities," *Proc. Design, Automation and Test in Europe Conf.* (DATE 07), ACM Press, 2007, pp. 1361-1366.
32. S. Yang and M.R. Greenstreet, "Simulating Improbable Events," *Proc. 44th Design Automation Conf.* (DAC 07), ACM Press, 2007, pp. 154-157.
33. A. Agiwal and M. Singh, "An Architecture and a Wrapper Synthesis Approach for Multi-clock Latency-Insensitive Systems," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design* (ICCAD 05), IEEE CS Press, 2005, pp. 1006-1013.
34. M.R. Greenstreet, "Implementing a STARI chip," *IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 95), IEEE CS Press, 1995, pp. 38-43.
35. A. Chakraborty and M.R. Greenstreet, "Efficient Self-Timed Interfaces for Crossing Clock Domains," *Proc. 9th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 03), IEEE CS Press, 2003, pp. 78-88.
36. J. Mekie et al., "Interface Design for Rationally Clocked GALS Systems," *Proc. 12th IEEE Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 06), IEEE CS Press, 2006, pp. 160-171.
37. L.R. Dennison, W.J. Dally, and D. Xanthopoulos, "Low-Latency Plesiochronous Data Retiming," *Proc. 16th Conf. Advanced Research in VLSI* (ARVLSI 95), IEEE CS Press, 1995, pp. 304-315.
38. C.K. Kwok, V.V. Gupta, and T. Ly, "Using Assertion-Based Verification to Verify Clock Domain Crossing Signals," *Proc. Design and Verification Conf.* (DVCon 03), MP Associates, 2003, pp. 18-26.

Paul Teehan is an MASc candidate in the Department of Electrical and Computer Engineering at the University of British Columbia, Vancouver. His research interests include high-performance interconnects, FPGA architectures, and networks on chips. Teehan has a BASc in computer engineering from the University of Waterloo, Ontario. He is a student member of the IEEE.

Mark Greenstreet is a professor in the Department of Computer Science at the University of British Columbia, Vancouver. His research interests include asynchronous design, high-performance interconnects, formal verification, and hybrid and dynamical systems. Greenstreet has a BSc in electrical engineering from the California Institute of Technology, and an MA and PhD in computer science from Princeton University. He is a member of the IEEE.

Guy Lemieux is an assistant professor in the Department of Electrical and Computer Engineering at the University of British Columbia, Vancouver. His research interests include FPGA architectures, CAD algorithms, and parallel computing. Lemieux has a BASc in engineering science, and an MASc and a PhD in electrical and computer engineering, all from the University of Toronto. He is a member of the IEEE and the ACM.

■ Direct questions and comments about this article to Paul Teehan, Dept of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada V6T 1Z4; pault@ece.ubc.ca.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.

Here now from the IEEE Computer Society

IEEE ReadyNotes

Looking for accessible tutorials on software development, project management, and emerging technologies? Then have a look at ReadyNotes, another new product from the IEEE Computer Society.

ReadyNotes are guidebooks that serve as quick-start references for busy computing professionals.

Available as immediately downloadable PDFs (with a credit card purchase), ReadyNotes sell for $19 or less.
www.computer.org/ReadyNotes

IEEE computer society