Fourteen Ways to Fool Your Synchronizer

Ran Ginosar
VLSI Systems Research Center, Technion—Israel Institute of Technology
Haifa 32000, Israel
[ran@ee.technion.ac.il]

Abstract

Transferring data between mutually asynchronous clock domains requires safe synchronization. However, the exact nature of synchronization sometimes eludes designers, and as a result synchronization circuits get "optimized" to the point where they do no longer operate correctly. This paper reviews a number of such cases, analyzes the causes of the errors, and offers a correct synchronizer circuit for each case. A correct two-flop synchronizer is presented. After discussing cases that avoid synchronization, the following synchronizers are reviewed: one flop, sneaky path, greedy path, wrong protocol, global reset, async clear, DFT leakage, pulse, slow-to-fast, metastability blocker, parallel and shared flop synchronizers.

1. Introduction

Transferring data between mutually asynchronous clock domains requires safe synchronization [1-6]. The operation of synchronization circuits has been recognized for a long time as being delicate and easy to disturb [1-3, 7-12], but at the same time robust synchronizer design does guarantee safe operation for all practical purposes. However, the exact nature of synchronization sometimes eludes designers, and as a result synchronization circuits get "optimized" to the point where they do no longer operate correctly. This paper reviews a number of such cases, analyzes the causes of the errors, and offers a correct synchronizer circuit for each case. The author has encountered those interesting cases while teaching, while working with various SOC (System on Chip) design teams, and while reviewing certain papers submitted for publication.

The paper starts by presenting a (hopefully) correct two-flop synchronizer. Validation means and tools are discussed. Section 3 describes the various synchronizers, analyzes the errors and pitfalls, and offers suggestions.

This paper focuses on the most general synchronization of two mutually-asynchronous clock domains. More aggressive synchronization circuits, which achieve high throughput data transfer between clock domains having the same or related frequencies, are not discussed here.

2. A Correct Two Flop Synchronizer

The simplest and safest method for the transfer of data between two mutually-asynchronous clock domains requires a *two-flop synchronizer* [2-4]. A "push" synchronizer is shown in Figure 1, but the principles apply also to pull, push-pull, and control-only synchronizers.

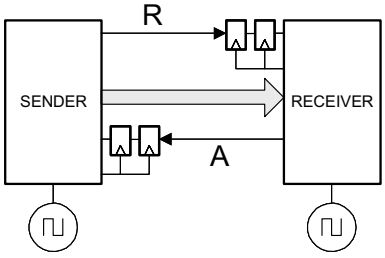

Figure 1: A push synchronizer

Bundled data is employed. The "synchronizer" actually comprises *two* synchronization circuits that envelope the data lines, implementing a complete handshake protocol. The Request (R) and Acknowledge (A) lines are synchronized by the receiver and sender, respectively. The settling window T (namely the time separation between the two clock inputs to the two flops of the synchronizers) could be a whole clock cycle or a fraction thereof, and could be different for each side, as long as the desired reliability is obtained. Synchronizer reliability is typically expressed in terms of Mean Time Between Failures [2]:

$$MTBF = \frac{e^{T/\tau}}{T_W f_A f_D}$$

where τ is the settling time constant of the flop, T_W a parameter related to its time window of susceptibility, f_A the synchronizer's clock frequency (the receiver's clock frequency for the R synchronizer and the sender's for the A synchronizer), and f_D is the frequency of pushing data across the clock domain boundary. Typically, MTBF is designed to be at least ten times the expected life of the

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 IEEE

product. If latency is not an issue, T is simply set to be a whole clock cycle, and for most SOCs it implies MTBF of many eons.

The two synchronizers connect two simple finite state machines that implement the required protocol. A four-phase protocol is specified by means of a generalized STG in Figure 2, where "DD" means that the data is available (at the sender), "UU" means that it may be removed, and "LL" means data latched by the receiver. (A two-phase protocol may also be employed; the circuits are a bit more complex [13, 14], and this is typically used in order to minimize latency on long lines.) The complete logic and FSM are shown in Figure 3. A send request (V, true for a single cycle) latches data into REG_S and starts the sender's FSM. The synchronized request (R2) latches the data into REG_R and triggers the receiver's FSM. The receiver is given a single-cycle "data received" (D) signal. The protocol is sometimes modified so that A is set as soon as the received data are latched, but removed only after the receiver has had an opportunity to use the data.

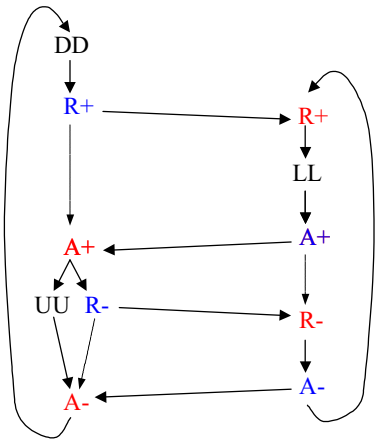

Figure 2: Four-phase handshake push synchronization protocol STG

To consider the synchronizer's behavior in cases of conflicts, assume that *T* equals a whole clock cycle. Upon a potential clock-data conflict on R, one of three possible outcomes may happen (Figure 4):

a. The rising edge of R is sampled high. R2 goes high on cycle 2, and data is latched into REG_R by the beginning of cycle 3.

b. The rising edge of R is sampled low. Since the protocol assures that R stays high as long is A is low, it will be sampled high on cycle 2, when it is surely stable high. R2 will go high on cycle 3, and data is latched into REG_R by cycle 4.

c. The first flop goes metastable. With a probability of $1-e^{-T/\tau}$ (which is infinitesimally close to 1), the flop has exited metastability by the next clock, and has arbitrarily settled to either high or low (the thick traces of R1 in the figure). If high, then R2 goes high on cycle 2. If low, it will surely go high on the next cycle, when the input R is already stable high, and R2 goes high on cycle 3.

A word of caution is due here: Although outcome c above implies that metastability typically disappears within a single clock cycle, the second flop is still required. An exception is discussed in Section 3.2 below.

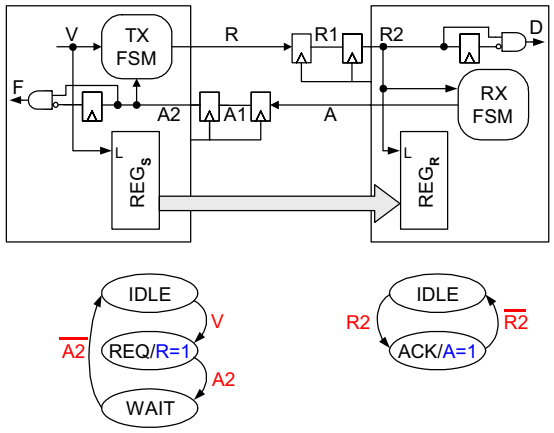

Figure 3: Push synchronizer logic and protocol FSM

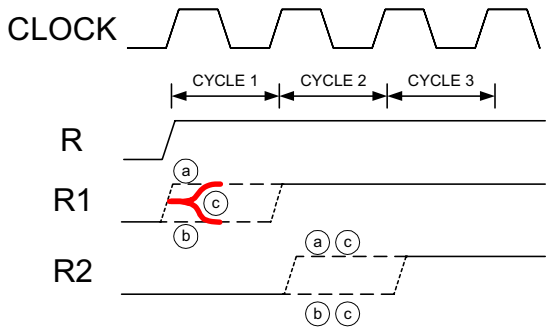

Figure 4: Three synchronization scenarios

A VHDL specification of the synchronizer is shown in Figure 5. This is a highly sensitive code, where minor modifications may render the synchronizer useless. Some such innovative but often fatal modifications are reviewed in the rest of this paper.

Logic validation tools are typically incapable of detecting any errors in such synchronizers. When reasonable logic assumptions are made, many erroneous synchronizers appear to operate perfectly well. Synchronizer-specific verification algorithms are required for this analysis.

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 IEEE

```
-- TRANSMITTER (inputs V, A, output R)
if rising_edge(tx_clock) then
        A2 <= A1; A1 <= A;                  -- 2 flop
        A3 <= A2; F <= not A3 and A2; -- 1 shot
        case (tx_fsm_state) is
          when idle =>
            if (V = '1') then
                tx_fsm_state <= req;
                R <= '1';
            end if;
          when req =>
            if (A2 = '1') then
                tx_fsm_state <= waiting;
                R <= '0';
            end if;
          when waiting =>
             if (A2 = '0') then
                tx_fsm_state <= idle;
             end if;
           when others =>
             tx_fsm_state <= idle;
             R <= '0';
        end case;
end if;

-- RECEIVER (input R, output A)
if rising_edge(rx_clock) then
        R2 <= R1; R1 <= R;                  -- 2 flop
        R3 <= R2; D <= not R3 and R2; -- 1 shot
        case (rx_fsm_state) is
          when idle =>
            if (R2 = '1') then
                rx_fsm_state <= ack;
                A <= '1';
            end if;
          when ack =>
            if (R2 = '0') then
                rx_fsm_state <= idle;
                A <= '0';
            end if;
          when others =>
             rx_fsm_state <= idle;
             A <= '0';
        end case;
end if;
```

Figure 5: Push 2-way 4-phase synchronizer VHDL specification

One tool has been developed specifically for validating synchronization. The Avant! Clock Domain Checker [15] is a decent first attempt at addressing this issue. However, it has a number of drawbacks: First, the control and data signals that cross domain boundaries must be named in a manner that facilitates these checks. Second, it validates only one-sided transfers and does not examine complete two-sided protocols and the protocol state machines. Third, it only validates a limited set of pre-defined rules, mostly covering a simple two-flop synchronizer and data lines protected by it; for instance, it does not check the synchronization of asynchronous reset. Fourth, it only handles "push" (and control-only) synchronizers, but neither "pull" nor "push-pull" ones. Another such tool is @Verifier from @HDL [16].

3. The Interesting Synchronizers

3.1 Avoiding the Synchronizer

The most common synchronization error is the transfer of a signal from one clock domain into another without any synchronization. In some cases the designer felt that failure probability was too low to worry about (he has learned about MTBF in the range of 10^{100} years, so why bother?). In other cases, the receiver operated at a much higher clock frequency than the sender, and the designer felt that the receiver would always be fast enough to catch the signal.

The incoming data is used as a combinational input to a combinational circuit, which eventually feeds into a flip-flop. Since the timing of the input is unknown, there is no way to guarantee the timing of the output of the combinational circuit. In particular, it may change simultaneously with the sampling edge of the clock, and the receiving flip-flop may enter metastability or take excessively long time to respond, hampering correct operation of the next stage of logic [2].

How often does the receiving flop enter metastability? The rate of entering metastability is $T_W \times f_D \times f_C$. For a 0.18µm SOC (where $T_W \approx 50$ps) with a clock domain operating at 200MHz and receiving data every 1000 cycles, that rate is 2000/sec, namely two metastability events every millisecond. Ignoring such a high rate does take some courage!

This error can sometimes evade detection by normal logic validation tools. Simulations may assume such timing relations among the different clocks that all timing constraints are met. Static timing analysis would generate setup and hold violation warnings for every signal that crosses domain boundaries, but due to the typically huge number of such warnings most designers treat them as chaff and ignore them, assuming that the synchronizers will handle all those issues anyway. Consequently, legitimate warnings can easily be overlooked.

The error can be detected by the following *clock domain crossing analysis*, which can be performed using standard path analysis, e.g. as offered by logic synthesizers and by static timing analyzers. All possible pairs of clocks must be identified. For each pair, the CAD tool is made to report all logic paths that begin in a flop driven by the first clock and end in a flop driven by the second clock. The resulting list should be studied, either manually or with an automated script, and every reported path must be approved. Typically, the crossing lists are carefully maintained and are used as 'false-path' specifications, instructing the analysis tool to ignore cross-domain paths that are already verified.

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 IEEE

3.2 One Flop Synchronizer

A deceptively effective means of cutting down on the two-flop synchronizer's latency is to remove one of the flops (Figure 6).

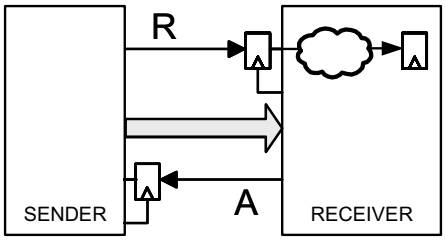

Figure 6: One-flop "synchronizer"

The problem comes about, of course, when there is a clock-data conflict. As explained above, the synchronizing flop may take an excessively long time to respond [2]. Its output may be used in a standard combinational logic stage (the cloud in the figure), whose nominal propagation delay is typically close to a whole clock cycle. When the synchronizing flop fails (responding slowly), the input to the next flop will not be ready in time for the next clock cycle.

The one-flop synchronizer can be detected by extending the analysis described above. The added step should validate that the output of every synchronizing flop feeds directly into the input of exactly one flop (driven by the same clock), without any logic in between.

The one-flop synchronizer is acceptable when designed correctly. If the delay through the combinational 'cloud' in Figure 6 is *d,* the settling time is T-*d.* If that time is sufficient to assure the required MTBF, then this synchronizer is legal.

3.3 Sneaky Path

Occasionally, a signal sneaks through a clock domain boundary unintentionally and unsynchronized. For instance, a signal has been moved from one clock domain to another as part of redesign, and some uses of the signal in its old domain are overlooked. It has also happened when a designer was unaware that a specific signal belonged to a different clock domain. In yet other cases, a signal *S* from a different clock domain is synchronized and renamed *S_sync*, but the designer has used *S* rather than *S_sync* by mistake.

The situation is similar to case 3.1 above, and so are the solutions.

3.4 Greedy Path Synchronizer

The designer employed a good two-flop synchronizer, but decided to save a little latency with the arrival detector: D= $R1\times\overline{R2}$ (Figure 7). This is quite similar to the one-flop synchronizer: The problem is that D is used with additional combinational logic, and the timing of that combinational path is typically designed to fit within a single clock cycle. But in cases of clock-data conflict of R, R1 may take longer than the normal flop t_{PD} to stabilize, and consequently the entire combinational path from R1 through D and to the last flop fails to converge during a single cycle. The right solution, obviously, is to add a flop and set D= $R2\times\overline{R3}$ (as in Figure 3).

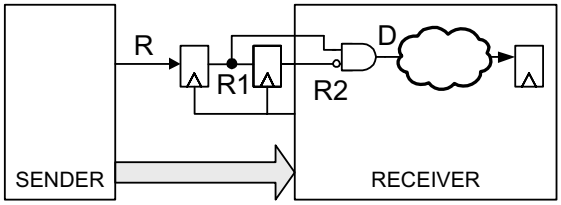

Figure 7: Greedy path "synchronizer"

3.5 Wrong Protocol

Consider the following example. The sender in a push synchronizer is a CPU that can be tuned to operate in the range of 60-100 MHz. The receiver is a communication modem based on a 55MHz clock. A push synchronizer is used to transfer data from the CPU to the modem. The designer has correctly realized that, once R is set, it would take at most four cycles of the receiver's clock to latch the data into REG_R (as in Figure 4). Based on the relative speeds, this would mean up to eight cycles of the faster sender's clock. To save time and logic, the designer eliminated the A line and its synchronizer; instead, he inserted a nine-cycle delay in the sender's FSM. After the delay, R is reset and the transfer is assumed finished.

There were two problems with that novel design. First, the designer did not realize that he had violated the safety (or 1-boundedness) requirement of the protocol (namely, transitions must be acknowledged, or else an STG arc might accumulate multiple tokens [17, 18]). Although the data was safely latched into REG_R, at times the receiver was busy doing something else and did not manage to make use of the data before a new set of data has arrived, over-writing the old.

Second, while the modem remained at 55MHz, the CPU in a later chip generation was sped up to 200MHz. At that rate, nine sender's clock cycles weren't enough any more to cover four modem cycles, and the synchronizer broke down.

There are other ways by which the protocol can be violated. A powerful protocol verification algorithm might provide a useful tool to weed out such innovations.

3.6 Global Reset Synchronizer

In a multi-frequency GALS (Globally Asynchronous, Locally Synchronous) SOC, a global reset signal is naturally asynchronous to at least some of the clock

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

domains. The leading edge of the reset signal is harmless, as it forces all circuits to a known starting state. The trailing edge, on the other hand, is the culprit in some chips. During global reset all the various clocks are started and all PLLs settle into their respective different frequencies. When the reset is removed, it can happen simultaneously with the sampling edge of one of the clocks. The global reset is typically connected into the asynchronous clear (or preset) input of many flip-flops, and its trailing edge must respect a setup constraint, or else the flops may enter metastability.

A safe interface is shown in Figure 8. It belongs with each of the several clock generators of the SOC. While the leading edge is transferred without delay (when the clocks may be inoperative), the trailing edge is synchronized.

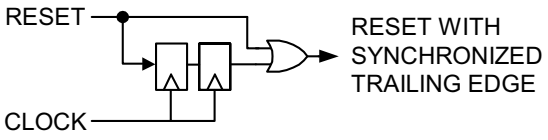

Figure 8: Global reset synchronizer

3.7 Async Clear Synchronizer

Occasionally (and contrary to the wisdom of typical synchronous design methodologies) asynchronous clear or preset of a flop may be employed as part of the logic (rather than for global reset, as discussed in Section 3.6). Some designers feel that, since this is an asynchronous clear, it needs not be synchronized even when it crosses clock domain boundaries (Figure 9).

The problem is very similar to that described in Section 3.6: Removal of the asynchronous clear signal may concur with the rising edge of the receiver's clock, potentially leading to metastability. The solution is either to synchronize the reset signal with two flops, or (when the leading edge must not be delayed) design an asymmetric synchronizer as in Figure 10.

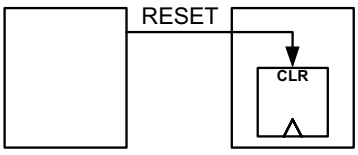

Figure 9: Asynchronous clear

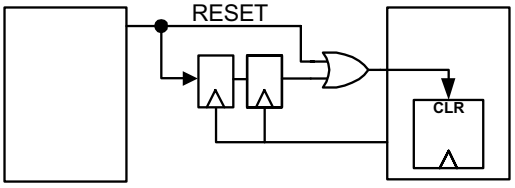

Figure 10: Synchronized-trail clear

3.8 DFT Leakage

Simple production testers may have only a single clock. To test a GALS SOC on such testers, all clocks are shorted together. Static faults (such as stuck-at) and some dynamic faults (speed testing of the individual clock domains) are properly tested that way. The clock shorts of course must be ignored during path analysis (by means of manually assembled 'false-path' lists or by instructing the analysis to ignore all paths that are conditioned upon a test-enable signal). But certain changes of the design may result in an error (sneaky) path masked by the list.

The solution is to recheck the entire false-path list as a final check, after all design changes are completed.

3.9 Pulse Synchronizer

The pulse synchronizer (Figure 11) is designed to pass a single "pulse" (a logic signal that is set to '1' for only a single clock cycle) from one clock domain to another. A pulse on P causes the sender's flop to toggle. Eventually, D is set high for a single cycle of the receiver's clock as a result.

The designer was lucky to discover the problem when the circuit was tried on an FPGA, prior to tapeout. Sometimes the P input was set to '1' for two consecutive cycles. At other times two pulses came in succession, with only one cycle in between. In both cases the synchronizer has generated undesirable results. The astute reader can easily figure out what they were. The situation was mended by replacing this with a standard control-only synchronizer, operating with a standard two—phase protocol.

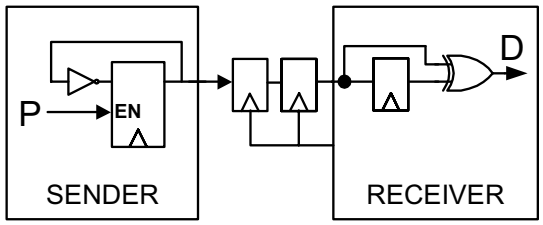

Figure 11: Pulse "synchronizer"

3.10 Slow-to-Fast Synchronizer

When the sender uses a slower clock than the receiver, designers can simplify the handshake protocol: The R line, when set for a single cycle of the sender's clock, is sampled by at least two edges of the receiver's clock. If the first edge misses, the second one is guaranteed to sample R. If the first one succeeds, further sampling is blocked (Figure 12), so that metastability during the second edge is avoided.

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

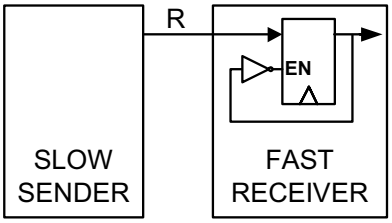

Figure 12: Slow-to-fast "synchronizer"

Such a simplified synchronizer typically works just fine. Except that SOCs tend to evolve and change clock frequencies. Sometimes clocks are changed during the design, when certain frequencies turn out to be too fast. In other cases, when a new product generation is launched or when the SOC is ported to a different fabrication process, slow domains may be sped up, and the assumption of who's faster may no longer hold. Hopefully the assumption has not been forgotten in the meantime, and the only adverse effect is that the chip needs to undergo a new logic and physical design, merely due to the 'optimized' synchronizer.

3.11 Metastability Blocker

A designer has suggested blocking metastability by the circuit of Figure 13. RESET clears the SR latch and the synchronizing flop. When the clock is high, if INPUT rises, the latch is set. When the clock goes low, the asynchronous input is blocked and only the SR latch output is connected to the flop. When the clock rises, it samples the synchronous output of the latch, rather than the asynchronous input.

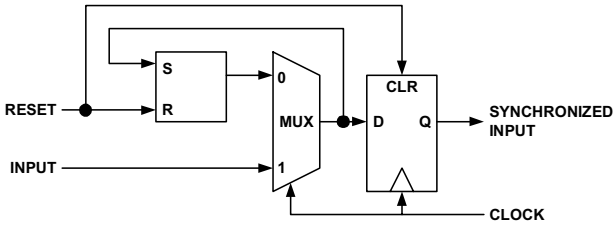

Figure 13: Metastability "blocker"

The designer has missed two problem scenarios, though. If INPUT rises exactly when the clock goes low, the SR latch can become metastable. It will most likely settle by the next rising edge of the clock. In other words, the metastability risk has simply been transferred from the flop to the latch, and one-half clock cycle is allowed for settling. If the proper protocol is employed (e.g., INPUT stays high until acknowledged), the synchronization will function correctly.

The second scenario is more dangerous. If INPUT rises exactly when the clock rises, the SR latch will probably miss it but the flop may become metastable. While the first scenario seems to be handled properly by the circuit (in spite of the designer's ignorance), the latter case may cause damage in the circuit that follows the flop.

Various "metastability blockers" or circuits that "eliminate" metastability are repeatedly reinvented and occasionally get published. Fortunately, most practitioners have learned to take them with a grain of salt.

3.12 Parallel Synchronizer

A careful designer assumed that more is better and, instead of using the recommended complex structure for a push synchronizer, he inserted a separate two-flop synchronizer on each data line (Figure 14). That scheme also seems to save one cycle time (no need to wait one full cycle after R2 is stable and until REG_R latches the incoming data, as in Figure 3).

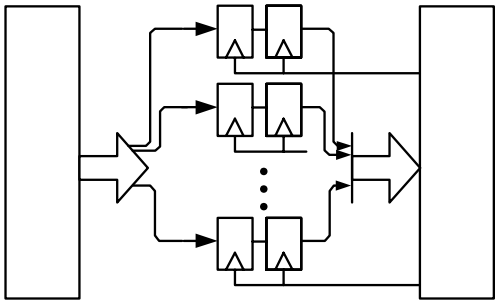

Figure 14: Parallel "synchronizer"

This scheme is a yet another prescription for a sure disaster. On clock-data conflict, each of the several data synchronizers may end up doing something different: Some may sample the new data, others may miss it and retain the old data, while yet others may enter metastability. Of the metastable ones, some may settle to '1' while others may settle to '0'. There is no way of telling which is which, as all four options are equally legitimate and possible outcomes.

To emphasize the severity of failure, recall that a typical single synchronizer may enter metastability twice every millisecond, as computed in Section 3.1. Thus, a 32 bit parallel synchronizer faces a risk of failure every 16 microseconds!

Another incarnation of this problem employs three parallel synchronizers and takes a vote of their outputs. Is this any safer than the non-voting parallel synchronizer?

3.13 Shared Flop Synchronizer

The synchronization handshake protocol is sometimes implemented with a signaling latch, set by the sender and cleared by the receiver. A somewhat misleading example based on two signaling flops has been published by a leading FPGA vendor (Figure 15). The problem is that the RECEIVE signal, which is driven by the sender's clock, is

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 IEEE

never synchronized by the receiver's (at least not in the schematics shown in the publication).

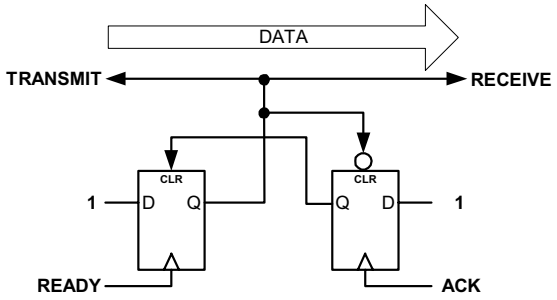

Figure 15: Shared flop "synchronizer"

A better scheme for a shared latch synchronizer (Figure 16) has been shown by Dike [19] and has been employed successfully in a low-voltage product (low supply voltage increases the risk of metastability). The control signals generated by the shared latch are both carefully synchronized with their respective clocks.

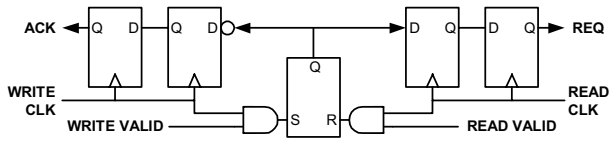

Figure 16: A correct shared latch synchronizer

3.14 Conservative Synchronizer

The careful designer occasionally wishes to be on the safe side and, when synchronization latency is not an issue, adds "just a few more stages" to the synchronizer (Figure 17). While this is not an error, it is interesting to learn what additional level of safety is thus obtained. Considering an SOC with two clock domains where the receiver operates at 200 MHz (a reasonable frequency for the 0.18μm technology), and where data is exchanged every ten clock cycles (as a worst case), and assuming T_W=50ps, τ=10ps (all 'conservative' numbers), the normal two-flop MTBF is $e^{500}/2\times10^5 = 10^{204}$ years. This is rather safe, when we recall that the age of the universe is 10^{10} years. The added cycle time provides an extra safety factor of e^{500}, achieving a more comforting level of 10^{420} years. Imagine how much better MTBF could have been if you used four flops, rather than three!

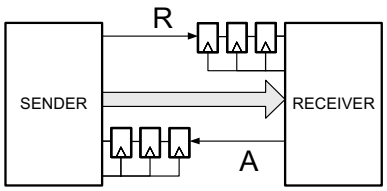

Figure 17: Conservative synchronizer

4. Conclusions

A few examples of synchronization design errors have been presented and analyzed. As long as there are no fool-proof algorithms and tools to validate synchronizers, the rules to safe design should be closely watched. A strict design methodology and discipline should be enforced, especially prohibiting arbitrary "improvements" of synchronizers and shortcuts in their design and implementation. Optimizations that may impede future design reuse should be avoided. Knowledgeable rigorous validation should be carried out to verify that all crossings of clock domains are understood and legitimate. Global signals that span multiple domains, such as reset and clocks, should be examined carefully. Such validation should be repeated after every design change and before final design closure.

Present efforts to design synchronizer cell libraries and to develop rigorous tools for synchronization validation may help alleviate these issues and assure safe GALS SOCs.

Synchronization issues may be more difficult to examine and validate with third-party IP cores, and especially "hard" cores whose internal logic design is unknown to the SOC designer. The architect should insist on at least a complete specification of their synchronizing circuits.

A certain type of synchronizers has not been dealt with in this paper, namely fast synchronizers for multi-sync [20] or mesochronous [4, 5] clock domains. Their design and validation are more complex and deserve another paper.

Acknowledgement

The author is grateful to the many imaginative designers whose innovations ended up in this paper. Their names are kept in confidence. The anonymous referees added some interesting examples to this catalog and helped weed out some of the bugs; the author alone should be blamed for any remaining mistakes.

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

References

[1] J. Jex and C. Dike, "A fast resolving BiNMOS synchronizer for parallel processor interconnect," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 133-139, 1995.

[2] C. Dike and E. Burton, "Miller and Noise Effects in a Synchronizing Flip-Flop," *IEEE Journal of Solid-State Circuits*, vol. 34, pp. 849-855, 1999.

[3] D. J. Kinniment, A. Bystrov, and A. Yakovlev, "Synchronization Circuit Performance," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 202--209, 2002.

[4] W. J. Dally and J. W. Poulton, *Digital System Engineering*(Eds.): Cambridge University Press, 1998.

[5] T. H.-Y. Meng, *Synchronization Design for Digital Systems*(Eds.): Kluwer Academic Publishers, 1991.

[6] D. J. Kinniment and J. V. Woods, "Synchronization and Arbitration Circuits in Digital Systems," *Proceedings of the IEE*, vol. 123, pp. 961--966, 1976.

[7] T. J. Chaney and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, vol. C-22, pp. 421--422, 1973.

[8] M. Pechoucek, "Anomalous Response Times of Input Synchronizers," *IEEE Transactions on Computers*, vol. 25, pp. 133--139, 1976.

[9] W. Fleischhammer and O. Dortok, "The anomalous behavior of flip-flops in synchronizer circuits," *IEEE Transactions on Computers*, vol. 28, pp. 273--276, 1979.

[10] H. J. M. Veendrick, "The Behavior of Flip-Flops Used as Synchronizers and Prediction of Their Failure Rate," *IEEE Journal of Solid-State Circuits*, vol. 15, pp. 169--176, 1980.

[11] L. Kleeman and A. Cantoni, "Can redundancy and masking improve the performance of synchronizers," *IEEE Transactions on Computers*, vol. 35, pp. 643--646, 1986.

[12] Y. Semiat and R. Ginosar, "Timing Measurements of Synchronization Circuits," under http://www.ee.technion.ac.il/~ran --> publications.

[13] P. Day and J. V. Woods, "Investigation into Micropipeline Latch Design Styles," *IEEE Transactions on VLSI Systems*, vol. 3, pp. 264--272, 1995.

[14] A. Peeters and K. v. Berkel, "Single-Rail Handshake Circuits," in *Asynchronous Design Methodologies*: IEEE Computer Society Press, 1995, pp. 53--62.

[15] "Clock Domain Checker User Manual," Avant! Corporation v2001.3, 2001.

[16] atHDL, "Multiple Clock Domain Analysis," www.athdl.com.

[17] A. V. Yakovlev, "On Limitations and Extensions of STG model for Designing Asynchronous Control Circuits," in *Proc. International Conf. Computer Design (ICCD)*: IEEE Computer Society Press, 1992, pp. 396--400.

[18] *Principles of Asynchronous Circuit Design: A Systems Perspective*, S. Furber (Eds.): Kluwer Academic Publishers, 2001.

[19] C. Dike, "Sychronization Tutorial," presented at Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC2000), 2000.

[20] R. Ginosar and R. Kol, "Adaptive Synchronization," in *Proc. International Conf. Computer Design (ICCD)*, 1998, pp. 188--189.

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**