Metastability and Synchronizers

A Tutorial

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

© IEEE 2011

Late Friday afternoon, just before locking the lab and leaving J to keep on churning cycles for the weekend, the sirens went off and all warning lights started flashing red. J, the new interplanetary spacecraft, has been undergoing full active system tests for a year now, without a glitch. But now, J's project managers realized, all they had were a smoking power supply, a dead spacecraft, and no chance of meeting the scheduled launch.

All the lab engineers and all J's designers could not put J back together again. They tried every test in the book, but could not figure out what happened. Finally, they called up K from the other side of the continent. It took him a bit, but eventually K managed to uncover the culprit: metastability failure in a supposedly-good synchronizer. The failure led the logic into an inconsistent state that turned on too many units simultaneously. That overloaded the power supply, which eventually blew up. Luckily, it happened in pre-launch tests and not a zillion miles away from Earth…

Metastability events are common in digital circuits, and synchronizers are a must to protect us from their fatal effects. Originally, they were required when reading an asynchronous input (namely, not synchronized with the clock so it might change exactly when sampled). Now, with multiple clock domains on the same chip, synchronizers are required when on-chip data cross the clock domain boundaries.

Any flip-flop can easily be made metastable. Toggle its data input simultaneously with the sampling edge of the clock, and you get it. One common way of demonstrating metastability is by supplying two clocks that differ very slightly in frequency to the data and clock inputs; every cycle the relative time of the two signals changes a bit, and eventually they switch sufficiently close to each other, leading to metastability. This coincidence happens repeatedly, enabling measurements with normal instruments.

Understanding metastability and the correct design of synchronizers is sometimes an art. Stories of malfunction and bad synchronizers are plenty. They cannot be synthesized, they are hard to verify, and often what has been good in the past may be bad in the future. Papers, patents and application notes giving wrong instructions are too numerous, as well as library elements and IP cores from reputable sources that might be 'unsafe at any speed.' This paper allows a glimpse into the theory and practice, and provides a short list of where to learn more about this fascinating subject.

1. Into and Out of Metastability

What is metastability? Consider the cross-cut through a vicious mini-golf trap in Figure 1. Hit the ball too lightly, and it remains where ball 1 is. Hit it too hard, and it reaches position 2. Can you make it stop and stay at the middle position? It is metastable, because even if your ball has landed and stopped there, the slightest disturbance (such as the wind) will make it fall to either side. And we cannot really tell to which side it will eventually fall.

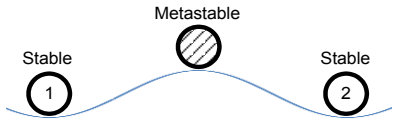

Figure 1: Mechanical metastability

In flip-flops, metastability means indecision of whether the output should be '0' or '1'. Here is a simplified circuit analysis model. The typical flip-flops in Figure 2 comprise master and slave latches and decoupling inverters. In metastability, the voltage levels of nodes *A,B* of the *master* latch are roughly mid-way between logic '1' (V_{DD}) and '0' (GND). Exact voltage levels depend on transistor sizing (by design, as well as due to arbitrary process variations) and are not necessarily the same for the two nodes. However, for sake of simplicity assume that they are (V_A=V_B=V_{DD}/2).

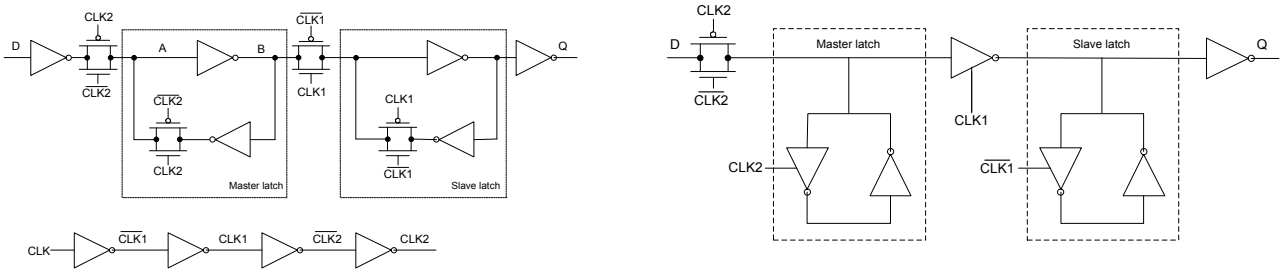

Figure 2: Two flip flops, with four (left) and two (right) gate delays from D to Q

1.1 Entering Metastability

How does the master latch enter metastability? Consider the flip-flop on the left in Figure 2. Assume that the clock is low, node *A* is at '1' and input *D* changes from '0' to '1'. As a result, node *A* is falling and node *B* is rising. When the clock rises, it disconnects the input from node *A* and closes the *A—B* loop. If *A* and *B* happen to be around their metastable levels, it would take them a long time to diverge away towards legal digital values, as shown below. In fact, one definition says that if the output of a flip-flop changes later than the nominal clock-to-Q propagation delay (t_{pCQ}) then the flip-flop must have been metastable. We can simulate the effect by playing with the relative timing of clock and data, until we obtain the desired result, as demonstrated in Figure 3. Incidentally, other badly timed inputs to the flip-flop (asynchronous reset, clear, and even too short a pulse of the clock due to bad clock gating) could also result in metastability.

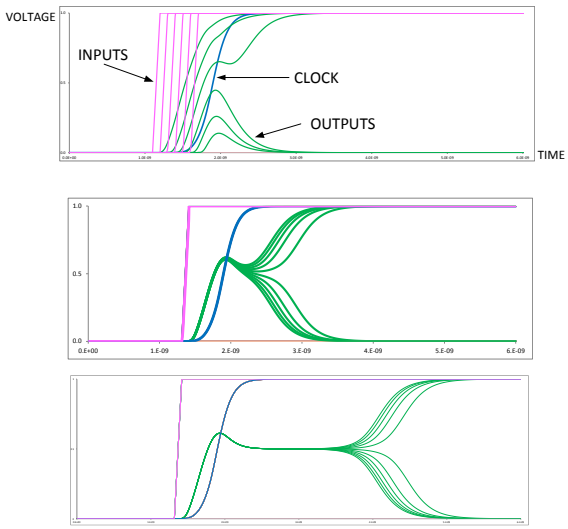

Figure 3: Empirical circuit simulations of entering metastability in the master latch of Figure 2 (left). Charts show multiple inputs D, internal clock (CLK2) and multiple corresponding outputs Q (voltage *vs.* time). The input edge is moved in steps of 100ps, 1ps and 0.1fs in the top, middle and bottom charts, respectively

When the coincidence of clock and data is unknown, we employ probability to assess how likely the latch is to enter metastability (let's focus on the master latch for now, and discuss the entire flip-flop later). The simplest model for asynchronous input assumes that data is likely to change at any time with uniform distribution. We can define a short window T_W around the sampling edge of the clock (sort of "setup and hold time") such that if data changes during that window the latch may become metastable (namely, the flip-flop output may change later than t_{pCQ}). If it is known that data has indeed changed sometime during a certain clock cycle, and since the occurrence of that change is uniformly distributed over the clock cycle T_C, the probability of entering metastability, which is the probability of D having changed within the T_W window, is $T_W/T_C=T_WF_C$. But D may not change every cycle; if it changes at a rate F_D, then the *rate* of entering metastability becomes *Rate*$=F_DF_CT_W$. For instance, if F_C=1GHz, F_D=100MHz and T_W=20ps, then *Rate*=2,000,000 times/sec. Indeed the poor latch enters metastability quite often, twice per microsecond or once every 500 clock cycles! Note how we traded probability for rate—we need that below.

1.2 Exiting Metastability

Now that we know how often a latch has entered metastability, how fast does the latch exit from it? In metastability the two inverters operate at their linear-transfer-function region and can be (small-signal) modeled as (negative) amplifiers (Figure 4), each driving (through its output resistance *R*) a capacitive load *C* comprising the input capacitance of the other inverter as well as any other external load connected to the node. Typically, the master latch becomes metastable and resolves before the second phase of the clock cycle; in rare cases, when the master latch resolves precisely half a cycle after the onset of metastability, the slave latch may enter metastability as a result (its input is changing exactly when its clock disconnects its input, and so on, repeating the above story about the master latch).

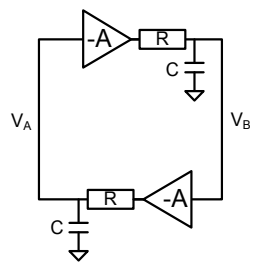

Figure 4 Analog model of a metastable latch; the inverters are modeled as negative amplifiers

This simple model results in two first-order differential equations that can be combined into one, as follows:

$$\frac{-AV_B - V_A}{R} = C\frac{dV_A}{dt}, \qquad \frac{-AV_A - V_B}{R} = C\frac{dV_B}{dt}$$

subtracting, $$\frac{V_B - V_A + A(V_A - V_B)}{R} = C\frac{d(V_A - V_B)}{dt}$$

define $V_A - V_B \equiv V$ and $\frac{RC}{A-1} \equiv \tau$

then $V = \tau\frac{dV}{dt}$ and the solution is $V = Ke^{t/\tau}$

Since $A/R \approx g_m$ we often estimate $\tau = C/g_m$; higher capacitive load on the master nodes or lower inverter gain impede the resolution of metastability. The master latch is exponentially sensitive to capacitance, and different latch circuits often differ mainly on the capacitive load they have to drive. While in the past τ was shown to scale nicely with technology, new evidence has recently emerged indicating that in future technologies τ may deteriorate rather than improve.

The voltage difference V thus demonstrates an "explosion" of sorts (like any other physical measure that grows exponentially fast, e.g., a chemical explosion). This behavior is best demonstrated by circuit simulation of a latch starting from a minute voltage difference V_0=1μV (Figure 5). The voltage curves of the two nodes do not appear to change very fast at all (let alone explode). However, observing the logarithm of the voltage difference V in Figure 6 reveals a totally different picture. The straight line from the initial voltage V_0 and up to about V_1=0.1V or $\log(V_1) = -1$ (V_1 is about the transistor threshold voltage, V_{TH}) traverses five orders of magnitude at an exponential growth rate, indicating that the "explosion" actually happens at the microscopic level. As the voltage difference approaches the transistor threshold voltage, the latch changes its mode of operation from two interconnected small-signal linear amplifiers (as in Figure 4) to a typical and slower digital circuit. We say that metastability has resolved as soon as the growth rate of V is no longer exponential (the log curve in Figure 6 flattens off).

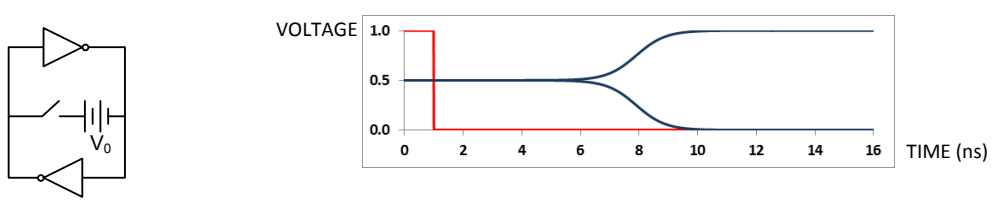

Figure 5: Simulation of exiting metastability: circuit (left) and voltage charts of the two latch nodes *vs.* time (right). The switch starts closed (applying V_0=1μV) and then opens up (at t=1ns) to allow the latch to resolve

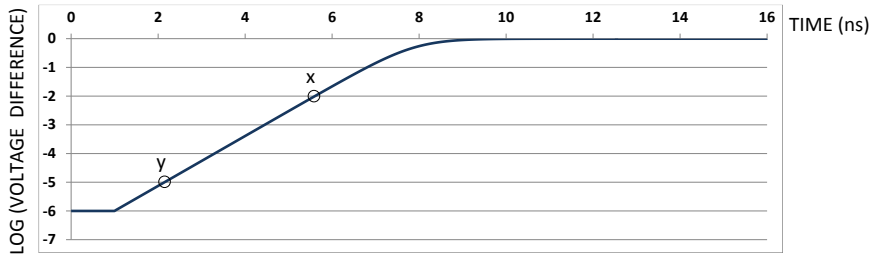

Figure 6: Log of the voltage difference of the two nodes of a resolving latch of Figure 5

The log chart facilitates a simple estimate of τ: take the ratio of two arbitrary voltage values $V_x(t_x)$, $V_y(t_y)$ along the straight line and solve for τ.

$$V_x = Ke^{t_x/\tau}\ ,\quad V_y = Ke^{t_y/\tau}$$

$$\frac{V_x}{V_y} = e^{(t_x - t_y)/\tau} \quad \Rightarrow \quad \tau = \frac{t_x - t_y}{\ln\left(V_x/V_y\right)}$$

Several factors affect τ in reality. First, in some circuits it may change during the resolution and the line of Figure 6 is not exactly straight. Process variations may result in τ several times larger than predicted by simulations. Low supply voltage, especially when the metastability voltage is close to the threshold voltage (and g_m decreases substantially), as well as very high or extremely low temperatures, may increase τ by several orders of magnitude! These issues make the research of synchronization an interesting challenge with practical implications.

Clearly, if metastability starts with $V=V_0$ and ends when $V=V_1$, then the time to exit metastability is t_m:

$$V_1 = V_0 e^{t_m/\tau} \quad \Rightarrow \quad t_m = \tau \ln\left(\frac{V_1}{V_0}\right)$$

Thus, the time to exit metastability depends logarithmically on the starting voltage V_0 (and not on the fixed exit voltage V_1), as clearly demonstrated by Figure 7.

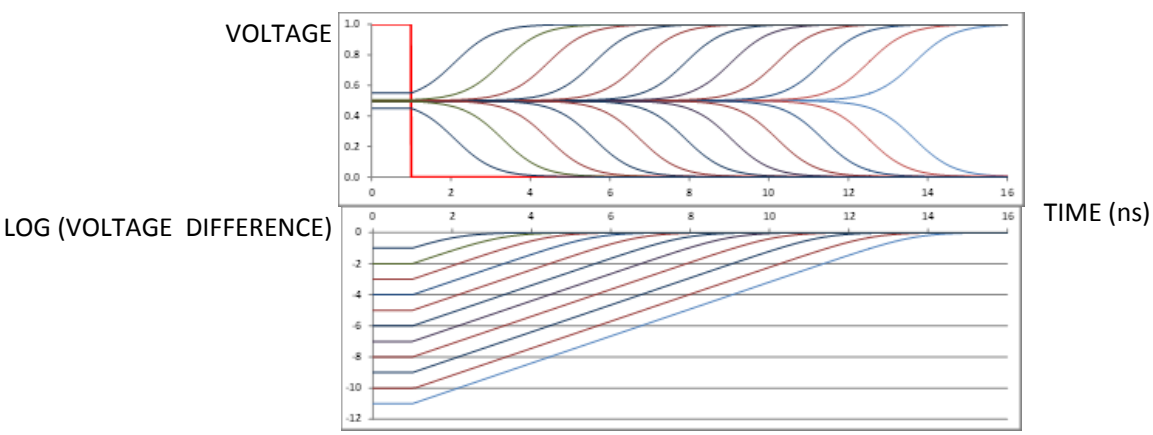

Figure 7: Simulations of metastability resolution with the starting voltage difference varying from 100mV (left) to 10pV (right); the lower the starting voltage, the longer it takes to resolve. The top chart shows

voltage of the two latch nodes (the chart for V_0=1μV is same as in Figure 5); the bottom chart shows the log of their difference (the line starting at −6 is the same as in Figure 6)

Had we started with $V_0=V_1$ then the time in metastability is zero (the leftmost curve in Figure 7). On the other hand, if V_0=0, we would have waited forever, but this is quite unlikely. In fact, the folklore stories claiming that 'in metastability, the two nodes of the latch get stuck in the middle and would eventually get out of there by some random process' should be taken lightly. The actual initial voltage is affected by two factors: when exactly did the sampling edge of the clock block the input and close the latch (namely what was the actual value of V at that moment), and how noise may have changed that value (observe that thermal noise is anywhere from 1μV to 1mV, much higher than V in the extreme cases on the right hand side of Figure 7). Since we do not know V_0 deterministically, we do not know how long the latch will stay metastable! But we can provide a statistical estimate. Probabilistic analysis shows that, given the fact that a latch is metastable at time zero, the probability that it will remain metastable at time t >0 is $e^{-t/\tau}$, which diminishes exponentially fast. In other words, even if a latch became metastable, it will resolve pretty fast!

1.3 Synchronization Reliability

All this leads us to computing reliability. If a latch receives asynchronous inputs, we cannot guarantee that it will never become metastable—in fact, we already know that it will definitely become metastable at the high rate of $F_D F_C T_W$ (2M/s in the example above)! Instead, we can compute the reliability that it will fail as a result. The whole purpose of the synchronizer (as discussed below) is to minimize that failure probability. So now we can finally define and estimate synchronization failures: We wish the metastability to resolve within a synchronization period S, so that we can safely sample the output of the latch (or flip-flop). Failure means that a flip-flop (*i*) has become metastable after the sampling edge of the clock, and (*ii*) it is still metastable S time later. The two events are independent, so we can multiply their probabilities:

$$p(failure) = p(enter\ MS) \times p(time\ to\ exit > S) = T_W F_C \times e^{-S/\tau}$$

Now we can take advantage of the expression for rate of entering metastability computed above to derive the rate of expected failures:

$$Rate(failures) = T_W F_C F_D \times e^{-S/\tau}$$

The inverse of the failure rate is the mean time between failures (MTBF):

$$MTBF = \frac{e^{S/\tau}}{T_W F_C F_D}$$

Let's design synchronizers with MTBF that is many orders of magnitude longer than the expected life time of the product. For instance, consider an ASIC designed for 28nm high performance CMOS process. We estimate τ=10ps, T_W=20ps (experimentally we know that both parameters are close to the typical gate delay of the process technology), and F_C=1GHz. Let's assume data changes every ten clock cycles at the input of our flip-flop, and we allocate one clock cycle for resolution: $S=T_C$. Plug all these into the formula and you obtain 4×10^{29} years. This is quite safe—the universe is believed to be only 10^{10} years old…

What happens at the flip-flop during metastability, and what can we see at its output? It's been said that we can see a wobbling signal that hovers around half V_{DD}, or that it can oscillate. Well, not exactly. If node A in Figure 2 is around $V_{DD}/2$, the chance that we can still see the same value at Q, three inverters later (or even one inverter, if the slave latch is metastable) is practically zero. Instead, the output will most likely be either '0' or '1', and as V_A resolves, the output may (or may not) toggle at some later time. If indeed that toggle happens later than the nominal t_{pCQ}, then we know that the flip-flop was metastable. And this is exactly what we want to mask with the synchronizer.

2. The Two-Flip-Flop Synchronizer

A simple two-flip-flop synchronization circuit is shown in Figure 8 (we don't call it a synchronizer yet—that comes later). The first flip-flop may become metastable. The second flip-flop samples Q1 a cycle later, hence $S=T_C$ (actually, any logic and wire delays are subtracted from the resolution time: $S=T_C-t_{pCQ}(FF1)-t_{SETUP}(FF2)-t_{PD}(wire)$, etc.). A failure means that

Q2 is unstable (it may change later than t_{pCQ}) and we know how to compute MTBF for that event. But what really happens inside there? Consider Figure 9, in which D1 switches dangerously close to the rising clock. Any one of six outcomes may happen:

(a) Q1 may switch at the beginning of cycle #1 and Q2 will copy that on clock cycle #2.
(b) Q1 may completely miss D1. It will surely rise on cycle #2, and Q2 will rise one cycle later.
(c) FF1 may become metastable, but its output stays low. It later resolves so that Q1 rises (the bold rising edge). This will happen before the end of the cycle (except, maybe, once every MTBF years). Then Q2 rises in cycle #2
(d) FF1 may become metastable, its output stays low, and when it resolves the output still stays low. This appears the same as case (b). Q1 is forced to rise in cycle #2 and Q2 rises in cycle #3.
(e) FF1 goes metastable, and its output goes high. Later, it resolves to low (we see a 'glitch' on Q1). By the end of cycle #1 Q1 is low. It rises in cycle #2, and Q2 rises in cycle #3.
(f) FF1 goes metastable, its output goes high, and it later resolves to high. Q1 appears the same as in case (a). Q2 rises in cycle #2.

The bottom line: Q2 is never metastable (except, maybe, once every MTBF years). Q2 goes high either one or two cycles later than the input. The synchronization circuit exchanges the 'analog' uncertainty of metastability for a simpler 'digital' uncertainty of whether the output switches one or two cycles later. Other than this uncertainty, the output signal is a solid legal digital signal.

What does happen when it really fails? Well, once every MTBF years, FF1 becomes metastable and resolves *exactly* one clock cycle later. Q1 may then switch exactly when FF2 samples it, possibly making FF2 metastable. Is this as unrealistic as it sounds? No. Run your clocks sufficiently fast, and watch for meltdown! Or you can go on reading and find out how to fight the odds.

A word of caution: the two flip-flops should be placed near each other, or else the wire delay between them would detract from the resolution time *S*. Missing this seemingly minor detail has made quite a few synchronizers fail unexpectedly.

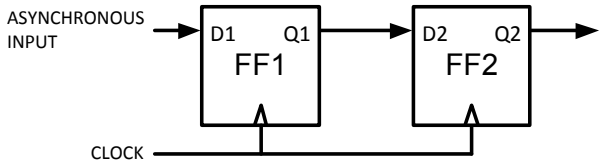

Figure 8: Two-flip-flop synchronization circuit

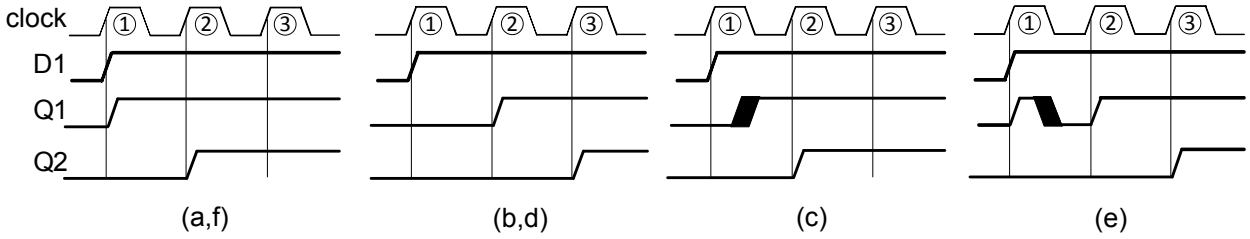

Figure 9: Alternative two-flip-flop synchronization waveforms

But this is only half the story. To assure correct operation, we assume in Figure 9 that D1 stays high for at least two cycles (in cases b,d,e) so that FF1 is guaranteed to sample '1' at its input on the rising clock of cycle #2. How would the sender know how long D1 must be kept high? We have no idea how fast the sender clock is ticking, so we cannot simply count cycles. To solve that, the receiver needs to send back an acknowledgement signal. Figure 10 (left) shows a *complete* synchronizer: The sender sends Req (a.k.a. Request, Strobe, Ready or Valid), Req gets synchronized by the top synchronization circuits, the receiver sends Ack (a.k.a. Acknowledgement, Stall), Ack gets synchronized by the sender, and only then is the sender allowed to change Req again. This round-trip handshake is the key to correct synchronization.

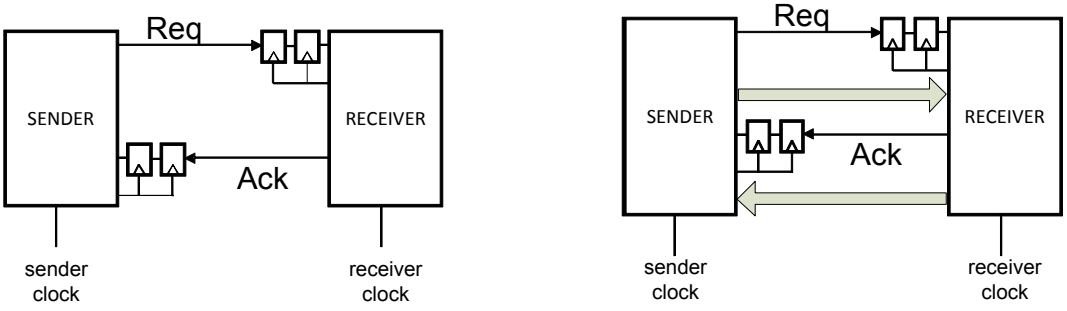

Figure 10: Complete two-way control synchronizer (left); complete two-way data synchronizer (right)

Now we can add data that need to cross over, as in Figure 10 on the right. The sender places data on the bus going to the right, and raises Req. Once the receiver gets wind of Req (synchronized to its clock), it stores the data and sends back Ack. It could also send back data on the bus going to the left. When the sender gets Ack, it can store the received data and also start a new cycle.

Note that the synchronizer does not synchronize the data. Rather, it synchronizes the control signals! Attempts to synchronize the data, bit by bit, usually lead to catastrophic results: even if all data lines toggle simultaneously, some bits may happen to pass through after one cycle while others may take two cycles, due to metastability. Beware, that's a complete loss of data. Another forbidden practice is to synchronize the same asynchronous input by two different parallel synchronizers: one may resolve to '1' while the other resolves to '0', leading to an inconsistent state. In fact, that was the problem that grounded the J spacecraft…

The two-flip-flop synchronizer comes in many flavors. When using slow clocks, resolution of less than half a cycle could suffice, as in the receiver side on the left hand side of Figure 11. In other cases, two flip-flops may not be enough. The clock may be fast (e.g., on processors that execute faster than 1GHz), the supply voltage may go very low (especially in near-threshold designs), and the temperature may either rise above 100ºC or drop way below freezing (have to use your phone outdoors on a cold Scandinavian winter night? or design a chip for the outside of an aircraft?). For instance, if $S=T_C$, F_C=1GHz, F_D=1kHz (we go conservative now), and due to low voltage and high temperature τ=100ps and T_W=200ps, MTBF is about one minute… Three flip-flops (sender side in Figure 11 on the right) would increase the MTBF a little to about one month. But if we use four flip-flops, $S=3T_C$ and the MTBF jumps to 1,000 years. Caution and careful design is the name of the game here!

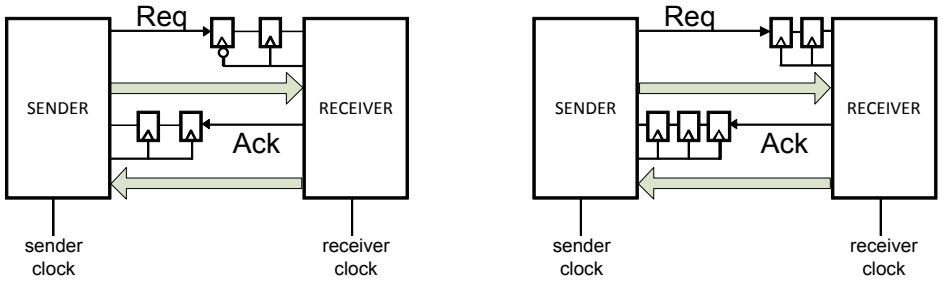

Figure 11: Variations on the theme of multi-flip-flops synchronization: half cycle resolution for a very slow clock (the receiver on the left) or three flip-flops enabling two cycles resolution for the fast clocks and extreme conditions (the sender on the right). As the sender and receiver may operate at widely different frequencies, different solutions may be appropriate.

Unique flip-flops designed especially for synchronization have been shown to be more robust to variations in process, voltage and temperature. Some employ current sources to enhance the inverter gain. Others sample multiple times and actively detect when synchronization is successful. The avid designer who has the freedom of using non-standard circuits

can take advantage of such inventions. The typical ASIC and FPGA designers are usually constrained to using only standard flip-flops, and will have to follow the well beaten path.

Another cause for concern is the total number of synchronizers in the design, be it a single chip or a system comprising multiple ASICs. MTBF decreases roughly linearly with the number of synchronizers. Thus, if your system employs 1000 synchronizers, make sure to design each one for at least three orders of magnitude higher MTBF than your reliability target for the entire system.

Similar concepts of synchronizations are employed for signals other than data that cross clock domains. Input signals may arrive at unknown timing. The trailing edge of the reset signal and of any asynchronous inputs to flip-flops are typically synchronized to each clock domain in a chip. Clock gating signals are synchronized to eliminate clock glitches when the clocks are gated or when a domain is switched from one clock to another. Scan test chains are synchronized when crossing clock domains. These applications are usually well understood and are well supported by special EDA tools for physical design.

The key issues, as usual, are latency and throughput. It may take two cycles of the receiver clock to receive R, two more cycles of the sender clock to receive A, and possibly one more on each side to digest its input and change state. If R and A need to be lowered before a new set of data can be transferred, consider another penalty of 3+3 cycles. No wonder we use F_D much lower than F_C in the examples above! This slow pace is fine for many cases, but occasionally we wish to work faster. Luckily, there are suitable solutions.

3. The Two-Clock FIFO Synchronizer

The most common fast synchronizer is the two-clock FIFO (Figure 12). Its advantages are hard to beat: you don't have to design it (it is usually available as a pre-designed library element or IP core), and it is (usually) fast. The writer places a data word on the input bus and asserts WEN; if FULL is not asserted, the data word was accepted and stored. The reader asserts REN, and if EMPTY is not asserted then a data word was produced at the output. The RAM is organized as a cyclic buffer. Each data word is written into the cell pointed to by the WRITE POINTER, and is read out when the READ POINTER reaches that word. On write and on read the WRITE POINTER and the READ POINTER are incremented, respectively.

When the READ POINTER points at the same word as the WRITE POINTER, the FIFO is empty. To determine that, the two pointers must be compared. However, they belong to two different clock domains. Thus, the WRITE POINTER has to be synchronized with RCLK when compared on the right. That's where the synchronization is: it is applied to the pointers, rather than to the data. That's also where latency is incurred: when a new data word is written into an empty FIFO, it might take one or two additional RCLK cycles before the new WRITE POINTER passes through the synchronizer and de-asserts EMPTY. But when the two pointers are far from each other, no synchronization latency is incurred; data latency is still there: when the RAM holds *k* words, a newly inserted word will stay there for at least *k* RCLK cycles before it is read out. Incidentally, the pointers are usually maintained in Gray codes so that only a single bit changes at a time in the synchronizer.

The FIFO solution usually works. It is non-trivial to design, but it is often available in libraries and elsewhere. The key question for the user of such a library FIFO is how large the RAM should be (how deep the FIFO is). The common approach says 'when in doubt, double it.' You might think that the life of the FPGA designer is simpler here: simply use trial and error. However, there have been cases where an FPGA failed in mission, due to too short a FIFO. Care is needed here.

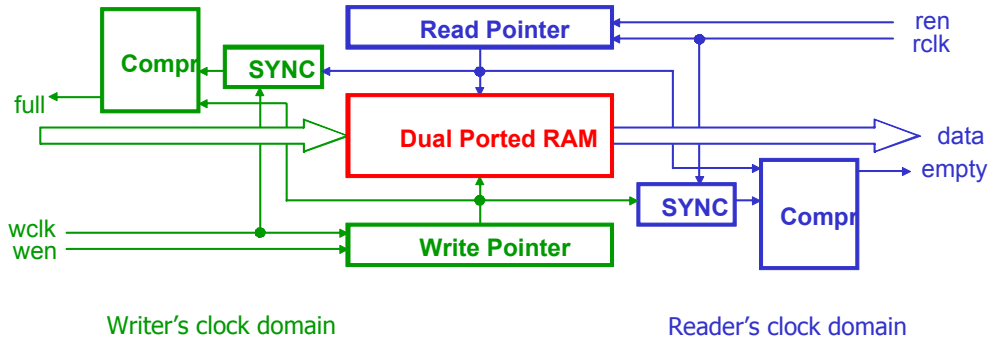

Figure 12: Two-clock FIFO synchronizer

The two-clock FIFO, as well as a mixed-timed FIFO interfacing a clock domain to an asynchronous (clock-less) circuits, are used in Networks-on-Chip (NoC) that offer connectivity among many modules on chip and also assume full responsibility for synchronization. This is a refreshing idea—let someone else (the NoC vendor or NoC designer) integrate your chip and take care of all clock domain crossings!

There are other fast synchronizers, which require higher design effort than simply using the common FIFO. The faster they are, the more complex the circuit. Most cases can be solved quite effectively with a good FIFO. Two special cases are discussed next: the mesochronous story and synchronizing over long distance.

4. Mesochronous, Multi-synchronous, Periodic and Rational Synchronizers

Two mesochronous clock domains tick to the same frequency, but their relative phase is unknown in advance. They are typically driven by the same clock, but no attempt is made to balance the two clock trees relative to each other (such balancing may incur heavy penalty in area and power). Once started, their relative phase remain stable. Figure 13 shows a common example: input X is sampled by each of the three registers in turn, and the oldest available sample is channeled to the output. The key question is how to set up the two counters, depending on the relative phase. The two-clock FIFO described above (with at least four stages) can also do the job: It should incur a one or two cycles synchronization latency at start-up, but thereafter the data latency is the same as in Figure 13. As an added advantage, the two-clock FIFO also enables back pressure: when the receiver stops pulling data out, the sender is signaled FULL and can stall the flow of data.

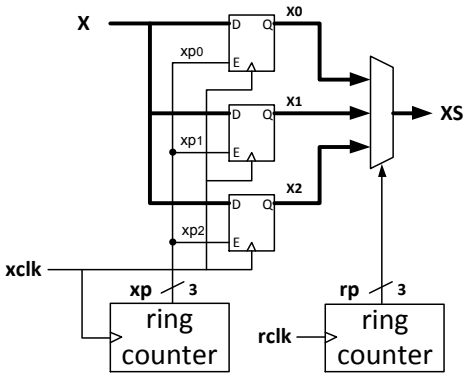

Figure 13: Mesochronous synchronizer

It turns out that mesochronous clock domains are not always mesochronous. The paths taken by a global clock to the various domains may suffer delay changes during operation, typically due to temperature and voltage changes. These drifts are typically slow, spanning many clock cycles. This may lead to domains operating at the same frequency but at slowly changing relative phases. Such relationship is termed *multi-synchronous,* to distinguish this case from mesochronous

operation. Synchronizers for multi-synchronous domains need to continuously watch out for phase drifts and adapt to them. A conflict detector (Figure 14) identifies when the sender and receiver clocks, XCLK and RCLK, are dangerously within one δ of each other (see the waveform on the right hand side of Figure 14); a useful value of δ is at least a few gate delays, providing a safe margin. The synchronizer (Figure 15) delays the clock of the first receiver register by t_{KO} ('keep-out' delay) if and only if XCLK is within δ of RCLK, as demonstrated by the waveform. This adjustment is made insensitive to any metastability in the conflict detector, because the phase drift is known to be slow. Typically the delay is changed only if the conflict detector has detected a change for a certain number of consecutive cycles, to filter out back-and-forth changes when XCLK hovers around RCLK ±δ. As above, the designer should also consider whether a simpler two-clock FIFO could achieve the same purpose. Incidentally, in addition to on-chip clock domain crossings, multi-synchronous domains can be found in phase-adaptive SDRAM access circuits and in clock / data recovery in circuits in high speed serial link SERDES systems.

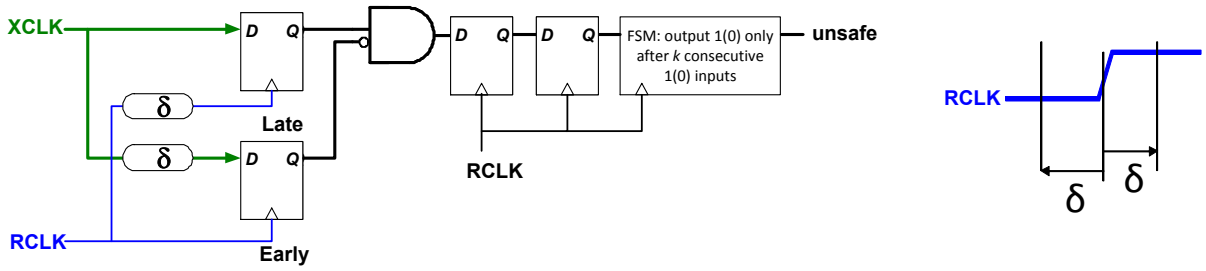

Figure 14: Conflict detector for multi-synchronous synchronization

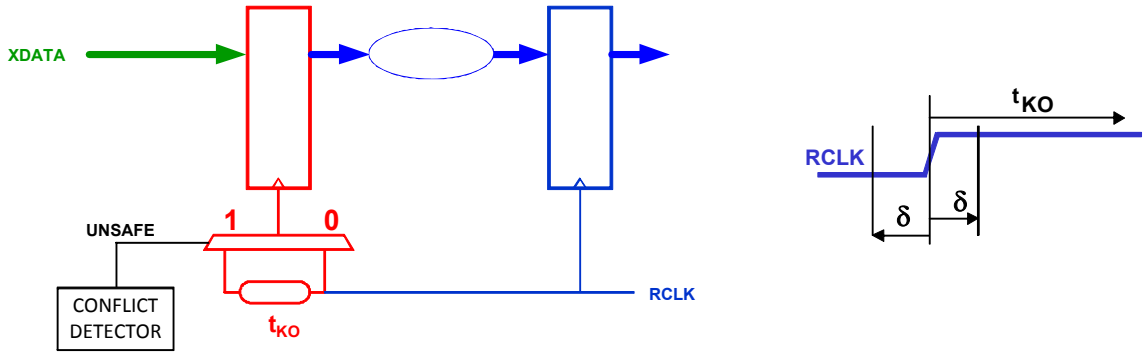

Figure 15: Synchronizer for multi-synchronous domains

A similar keep-out mechanism could be applied when synchronizing *periodic* clock domains. Periodic clocks are unrelated to each other—they are neither mesochronous nor are their frequencies an integral multiple of each other. Hence, we can expect that every few cycles the two clocks may get dangerously close to each other. But the conflict detector of Figure 14 is too slow to detect this on time (it may take k+2 cycles to resolve and produce the UNSAFE signal). Luckily, since the clock frequencies are stable, we can predict such conflicts in advance. A number of predictive synchronizers have been proposed but they tend to be complex, especially in light of the fact that the two-clock FIFO synchronizer may be suitable.

Another similar situation is that of *rational* clocks, where the two frequencies are related by a ratio that is known at design time (e.g., 1:3 or 5:6). In that case, determining danger cycles is simpler than in the case of periodic clocks with unknown frequencies, and a simple logic circuit could be used to control the clock delay selector of Figure 15.

Summarizing this section, we notice that different situations call for specific synchronizers that may excel on certain design parameters, but the conservative designer may often opt for the simpler, safer and commonly available two-clock FIFO synchronizer introduced in Sect. 3.

5. Long distance Synchronization

What is a long distance and what does it have to do with synchronizers? When we need to bridge the frequency gap between two clock domains placed so far apart that the signals take close to a clock cycle or even longer to travel between them, we face a new risk. The designer cannot rely on counting cycles when waiting for the signal to arrive—PVT variations (process, voltage and temperature) as well as design variations (actual floor plan, placement and routing) may result in an unknown number of cycles for traversing the interconnecting wires.

The simplest (and slowest) approach is to stretch the simple synchronizer of Figure 10 over the distance. It is slow because, when using return-to-zero signaling on Req, four 'flight times' over the distance are required before the next data word can be sent. We should guarantee, e.g., by means of timing constraints, that when Req has been synchronized (and the receiver is ready to sample its data input) the data word has already arrived; this is not trivial when Req and the data wires get routed through completely different areas of the chip. This safety margin requirement usually results in even slower operation.

The use of fast asynchronous channels help to somewhat mitigate the performance issue. Data bits are sent under the control of proper handshake protocols, and get synchronized when reaching their destination. The downside is the need for special IP cores, since asynchronous design is rarely practiced and is not really supported by common EDA tools.

While the latency is bounded by the physical distance, throughput over long channels may be increased by turning them into pipelines. But multi-stage pipelines require clocks at each stage, and it is not clear which clocks should be used in a multi-clock domain chip: when the data that are sent from Clock1 to Clock10 are routed near the areas of Clock2, Clock3, …, Clock9, all unrelated to either Clock1 or Clock10, which clocks do we use along the road? Some designs have solved it simply by clocking the pipes at the fastest frequency available on chip, but that solution is power-hungry.

The ultimate solution may lie in employing a network on chip (NoC). The network infrastructure is intended to facilitate multiple transfers among multiple modules, over varying distances, supporting varying clock frequencies. Asynchronous and synchronizing NoCs have been devised to address these issues and especially to provide complete synchronization while interfacing each source and destination module.

6. Verification

Since the design of proper synchronization is such an elusive goal, verification is a must. But for the same reason, verification is hard and unfortunately it does not always guarantee a correct solution.

Circuit simulations, such as shown in Sect. 1 (Figure 3), are useful for analyzing a single synchronizer but ineffective in proving that many synchronizations in a large SoC would all work correctly. An interesting simulation-based verification method has been developed at the logic level; recall from Sect. 2 that a good synchronizer should contain all level and timing uncertainties, replacing them by the simpler logic uncertainty of when a signal crosses over—it could happen in one cycle or in the next one. Assuming that we have selected a good synchronizer, logic verification of the rest of the system is facilitated by replacing the synchronizer with a special synchronous delay block, which inserts a delay of either k or $k+1$ cycles at random. While this approach leads to a design space of at least 2^n cases if there are n synchronization circuits, it is still widely used and quite effective in detecting many logic errors (but not all of them—it would not have helped the J spacecraft designers…).

There are several EDA verification software tools, commonly dubbed 'clock domain crossing checkers' or CDC for short, which identify and check all signals that cross from one domain to another. They employ a variety of structural design rules and are quite helpful in assuring that no such crossing remains unchecked. Some of them assess the overall SoC reliability, in terms of MTBF. At least one such tool also suggests solutions for problematic crossings in terms of synchronizer IP cores.

7. Summary

This short tutorial attempts to present both the beauty and the criticality of the subject of metastability and synchronizers. Many researchers and practitioners, over more than 65 years, have shared the excitement of trying to crack this tough nut. The elusive physical phenomena, the mysterious mathematical treatment, and the challenging engineering solutions have all contributed to making this field so attractive. The folklore, the myths, the rumors and the horror stories have added a

fun aspect to an area that is blamed for the demise of several products and for the large financial losses that resulted. Luckily, clear understanding of the risks, respect for the dangers, and a strict engineering discipline help us to avoid the pitfalls and create safe, reliable and profitable digital systems and products.

8. Acknowledgements

Chuck Seitz wrote the seminal Chapter 7 of the VLSI bible and got me started on this. The late Charlie Molnar pointed at the science of metastability, and Peter Alfke stressed the engineering of it. The comments of Charles Dike, Cliff Cummings, Shalom Bresticker, Shlomi Beer, Reuven Dobkin and Richard Katz helped to significantly improve this writing. The cumulative experience and know-how of hundreds of engineers, students and colleagues taught me the good and the bad and convinced me that this is a delightful subject!

9. Literature

Synchronizers were used as early as Eckert and Mauchly's ENIAC in the 1940s, but the first mathematical analysis was published in 1952 [1], and the first experimental reports of metastability appeared in 1973 [2]. Other early works and insight included [3]-[6]. An influential tutorial was published in 1987 [7]. Two books [8][9] and two chapters in a third book [10] were published on these topics. Synchronizers are analyzed in [11]-[13] and metastability measurements are reported in [14]-[16]. The latter also observes that metastability parameters such as τ may actually deteriorate with future scaling, renewing concerns of proper synchronizer design. Various methods for simulation of synchronizers are described in [11],[12],[17],[18]. Practical methods of creating two-clock FIFO synchronizers are detailed in [19], and mixed synchronous/asynchronous FIFO, e.g., for NoC, is proposed in [20]. Fast synchronizers are described in [21],[22], while [23] and [24] present synchronizers that are robust to extreme conditions, improving on a patent idea [25]. An early version of multi-synchronous synchronization is discussed in [26]. Errors and misperceptions of synchronizers are described in [27]. Examples of formal verification of synchronizers are presented in [28]-[29]. Finally, employing a NoC to solve all synchronization issues is discussed in [30].

[1] S. Lubkin, "Asynchronous Signals in Digital Computers," Mathematical Tables and Other Aids to Computation (ACM section), 6(40):238-241, 1952.
[2] T. J. Chaney and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," IEEE Trans. Comp., C-22:421—422, 1973.
[3] D. J. Kinniment and J. V. Woods, "Synchronization and Arbitration Circuits in Digital Systems," Proceedings of the IEE, 123:961--966, 1976.
[4] H. J. M. Veendrick, "The Behavior of Flip-Flops Used as Synchronizers and Prediction of Their Failure Rate," IEEE Journal of Solid-State Circuits, 15:169--176, 1980.
[5] M. Stucki, J. Cox, "Synchronization Strategies," Caltech Conf. on VLSI, 1979.
[6] C. Seitz, System Timing, Ch. 7 of C. Mean and L. Conway, Introduction to VLSI Systems, Addison-Wesley 1979.
[7] L. Kleeman and A. Cantoni, "Metastable Behavior in Digital Systems," IEEE Design & Test of Computers, 4:4-19, 1987.
[8] T. H.-Y. Meng, Synchronization Design for Digital Systems, Kluwer Academic Publishers, 1991.
[9] D. J. Kinniment, Synchronization and Arbitration in Digital Systems, Wiley, 2008.
[10] W. J. Dally and J. W. Poulton, Digital System Engineering, Cambridge University Press, 1998.
[11] C. Dike and E. Burton, "Miller and noise effects in a synchronizing flip-flop," IEEE Journal of Solid-State Circuits, 34:849-855, 1999.
[12] D. J. Kinniment, A. Bystrov, and A. Yakovlev, "Synchronization Circuit Performance," IEEE Journal of Solid-State Circuits, 37:202--209, 2002.
[13] S. Yang, M. Greenstreet, "Computing synchronizer failure probabilities," Proc. Design Automation and Test in Europe (DATE), pp. 1 – 6, 2007.
[14] L.-S. Kim, R. Cline, R.W. Dutton, "Metastability of CMOS latch/flip-flop," Proc. Custom Integrated Circuits Conference (CICC), pp. 26.3/1 - 26.3/4, 1989.
[15] Y. Semiat and R. Ginosar, "Timing Measurements of Synchronization Circuits," Proc. IEEE Int. Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 68 – 77, 2003.
[16] S. Beer, R. Ginosar, M. Priel, R. Dobkin and A. Kolodny, "The Devolution of Synchronizers," Proc. IEEE Int. Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 94 – 103, 2010.

[17] J. Jones, S. Yang, M. Greenstreet, "Synchronizer behavior and analysis," Proc. IEEE Int. Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 117 – 126, 2009.
[18] S. Yang, M. Greenstreet, "Simulating improbable events," Proc. 44th Design Automation Conf. (DAC 07), pp. 154–157, 2007.
[19] Cliff Cummings, "Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog," Proc. Synopsys User Group Meeting (SNUG), 2008 (http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf).
[20] T. Chelcea and S.M. Nowick, "Robust Interfaces for Mixed Timing Systems," IEEE Transactions on Very large Scale Integration (VLSI) Systems, 12(8):857—873, 2004.
[21] R. Dobkin, R. Ginosar and C. Sotiriou, "High Rate Data Synchronization in GALS SoCs," IEEE Transactions on Very large Scale Integration (VLSI) Systems, 14(10):1063-1074, Oct. 2006.
[22] R. Dobkin and R. Ginosar, "Two phase synchronization with sub-cycle latency," INTEGRATION, the VLSI journal, 42(3):367-375, 2009.
[23] J. Zhou, D. Kinniment, G. Russell and A. Yakovlev, "A Robust Synchronizer," Proc. IEEE Symposium on Emerging VLSI Technologies and Architectures (ISVLSI), 2006.
[24] M. Kayam, R. Ginosar and C.E. Dike, "Symmetric Boost Synchronizer for Robust Low Voltage, Low Temperature Operation," EE Tech. Rep., Technion, 2007.
[25] R. Cline, "Method and circuit for improving metastable resolving time in low-power multi-state devices", US Patent 5789945, 1998.
[26] R. Ginosar and R. Kol, "Adaptive Synchronization," Proc. Int. Conference on Computer Design (ICCD), 1998.
[27] R. Ginosar, "Fourteen Ways to Fool Your Synchronizer," Proc. IEEE Int. Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 89 – 96, 2003.
[28] U. Frank, T. Kapschitz and R. Ginosar, "A Predictive Synchronizer for Periodic Clock Domains," J. Formal Methods in System Design, 28(2):171-186, 2006.
[29] R. Dobkin, T. Kapshitz, S. Flur and R.Ginosar, "Assertion Based Verification of Multiple-Clock GALS Systems," Proc. IFIP/IEEE Int. Conference on Very Large Scale Integration (VLSI-SoC), 2008.
[30] E. Beigné, F. Clermidy, P. Vivet, A. Clouard and M. Renaudin, "An Asynchronous NOC Architecture Providing Low Latency Service and its Multi-level Design Framework," Proc. IEEE Int. Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 54 – 63, 2005.

Short author bio: Ran Ginosar, professor at the Electrical Engineering and Computer Science departments of the Technion—Israel Institute of Technology. He received his PhD from Princeton University and has since visited with, consulted for and co-founded multiple companies, where he learned the art, science, engineering and practice of synchronization. He conducts research on VLSI architecture.