

Metastability and Synchronizers: A Tutorial

Ran Ginosar

Technion—Israel Institute of Technology

■ **METASTABILITY EVENTS ARE** common in digital circuits, and synchronizers are a necessity to protect us from their fatal effects. Originally, synchronizers were required when reading an asynchronous input (that is, an input not synchronized with the clock so that it might change exactly when sampled). Now, with multiple clock domains on the same chip, synchronizers are required when on-chip data crosses 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 metastability. One common way to demonstrate metastability is to supply two clocks that differ very slightly in frequency to the data and clock inputs. During 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 demonstration of metastability with normal instruments.

Understanding metastability and the correct design of synchronizers to prevent it is sometimes an art. Stories of malfunction and bad synchronizers are legion. Synchronizers cannot always 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 article offers a glimpse into the theory and practice of metastability and synchronizers; the sidebar “Literature Resources on Metastability” provides a short list of resources where you can learn more about this subject.

Late Friday afternoon, just before the engineers locked the lab to leave J, the new interplanetary spacecraft, churning cycles all weekend, the sirens went off and the warning lights started flashing red. J had been undergoing fully active system tests for a year, without a single 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 couldn’t figure out what had happened. Finally, they phoned K, an engineering colleague on the other side of the continent. It took him a bit of time, but eventually K uncovered the elusive culprit: metastability failure in a supposedly good synchronizer. The failure led the logic into an inconsistent state, which turned on too many units simultaneously. That event overloaded the power supply, which eventually blew up. Luckily, it happened in prelaunch tests and not a zillion miles away from Earth.

Into and out of metastability

What is metastability? Consider the crosscut through a vicious miniature-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

Editors’ note:

Metastability can arise whenever a signal is sampled close to a transition, leading to indecision as to its correct value. Synchronizer circuits, which guard against metastability, are becoming ubiquitous with the proliferation of timing domains on a chip. Despite the critical importance of reliable synchronization, this topic remains inadequately understood. This tutorial provides a glimpse into the theory and practice of this fascinating subject.

—Montek Singh (UNC Chapel Hill) and Luciano Lavagno (Politecnico di Torino)

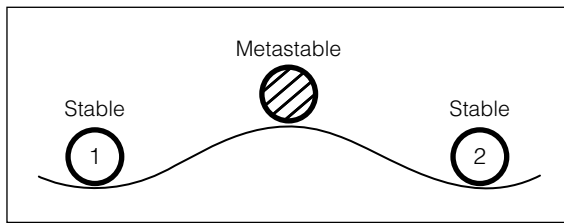


Figure 1. Mechanical metastability: the ball in the center position is “metastable” because the slightest disturbance will make it fall to either side.

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.

In flip-flops, metastability means indecision as to whether the output should be 0 or 1. Let’s consider a simplified circuit analysis model. The typical flip-flops comprise master and slave latches and

decoupling inverters. In metastability, the voltage levels of nodes A and B of the master latch are roughly midway 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 the sake of simplicity, assume that they are ($V_A = V_B = V_{DD}/2$).

Entering metastability

How does the master latch enter metastability? Consider the flip-flop in Figure 2a. 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 toward legal digital values, as Figure 3 shows. In fact, one popular definition says that if the output of a flip-flop changes later than the nominal clock-to-Q propagation delay

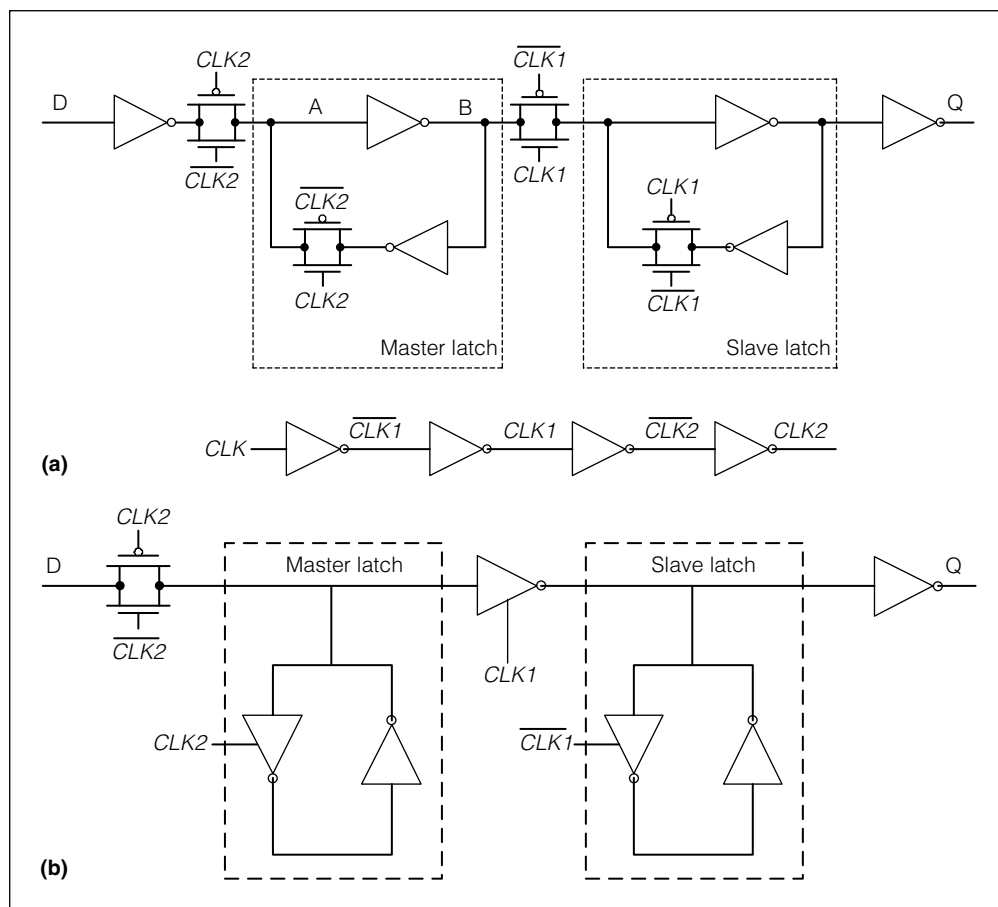


Figure 2. Flip-flops, with four (a) and two (b) gate delays from D to Q.

(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 Figure 3 demonstrates. 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.

When the coincidence of clock and data is unknown, we use probability to assess how likely the latch is to enter metastability (we 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 clock's sampling edge (sort of "setup-and-hold time") such that if data changes during that window, the latch could become metastable (namely, the flip-flop output might 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 clock cycle T_C —the probability of entering metastability, which is the probability of D's having changed within the T_W window, is $T_W/T_C = T_W F_C$ (F_C is the clock frequency).

But D may not change every cycle; if it changes at a rate F_D , then the rate of entering metastability becomes $Rate = F_D F_C T_W$. For instance, if $F_C = 1$ GHz, $F_D = 100$ MHz, and $T_W = 20$ ps, then $Rate = 2,000,000$ times/sec. Indeed, the poor latch enters metastability often, twice per microsecond or once every 500 clock cycles. (Note that we traded probability for rate—we need that in the following discussion.)

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 modeled, using small-signal analysis, as negative amplifiers (see Figure 4). Each inverter drives, through its output resistance R , a capacitive load C comprising the other inverter's input capacitance 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

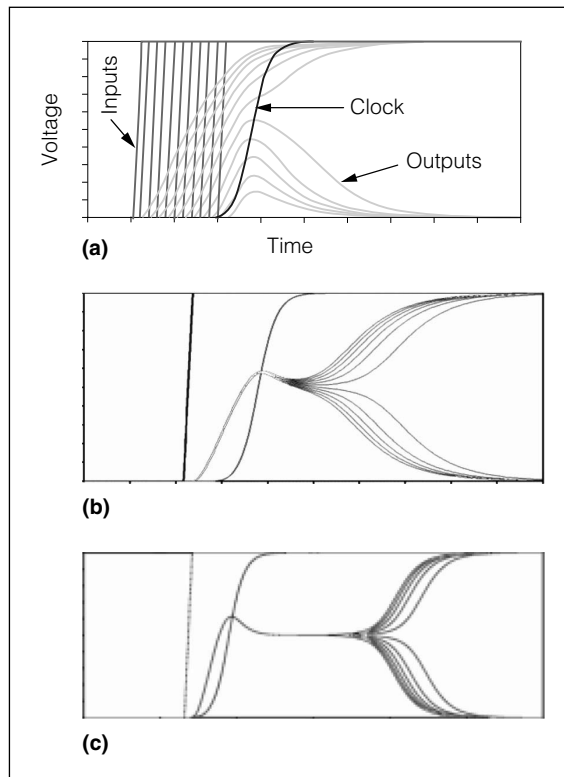


Figure 3. Empirical circuit simulations of entering metastability in the master latch of Figure 2a. Charts show multiple inputs D, internal clock (CLK2) and multiple corresponding outputs Q (voltage vs. time). The input edge is moved in steps of 100 ps, 1 ps, and 0.1 fs in the top, middle, and bottom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

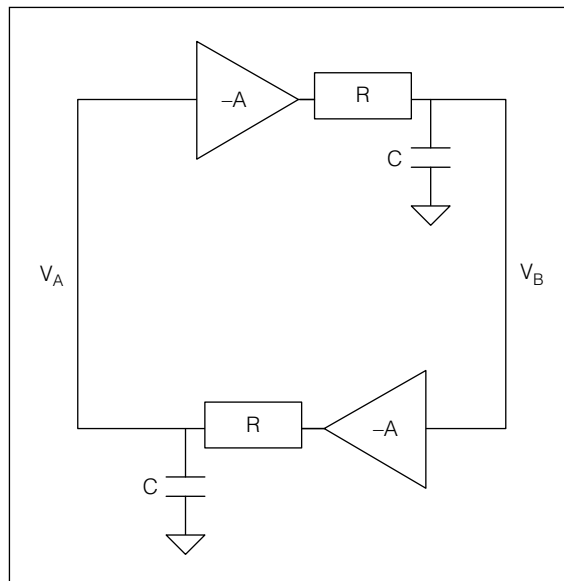


Figure 4. Analog model of a metastable latch; the inverters are modeled as negative amplifiers.

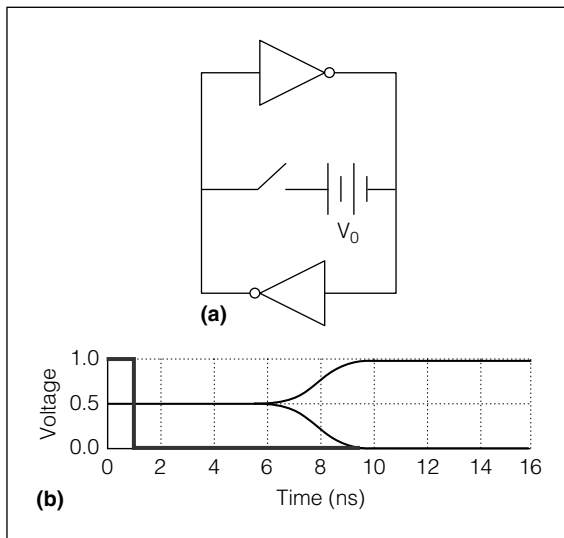


Figure 5. Simulation of exiting metastability: circuit (a) and voltage charts of the two latch nodes vs. time (b). The switch starts closed (applying $V_0 = 1 \mu\text{V}$) and then opens up (at $t = 1 \text{ ns}$) to allow the latch to resolve.

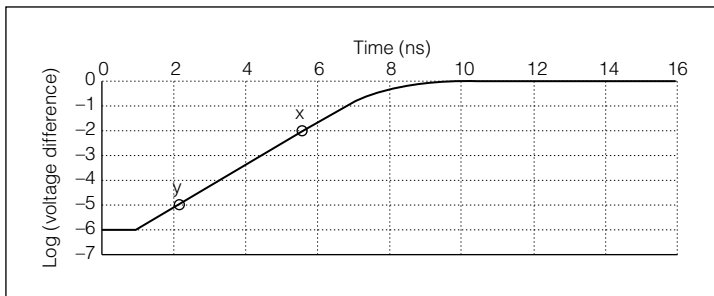


Figure 6. Log of the voltage difference of the two nodes of a resolving latch in Figure 5.

could enter metastability as a result (its input is changing exactly when its clock disconnects its input, and so on, thereby repeating the aforementioned master-latch scenario).

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

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

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

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

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

Because $A/R \approx g_m$, we often estimate $\tau = C/g_m$; higher capacitive load on the master nodes and 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. In the past τ was shown to scale nicely with technology, but 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., as in a chemical explosion). This behavior is best demonstrated by circuit simulation of a latch starting from a minute voltage difference $V_0 = 1 \mu\text{V}$ (see 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 up to about $V_1 = 0.1 \text{ V}$ or $\log(V_1) = -1$ (V_1 is approximately 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, 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).

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 :

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

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

Actually, several factors affect τ . First, in some circuits it may change during resolution, and the line of Figure 6 is not exactly straight. Process variations might 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

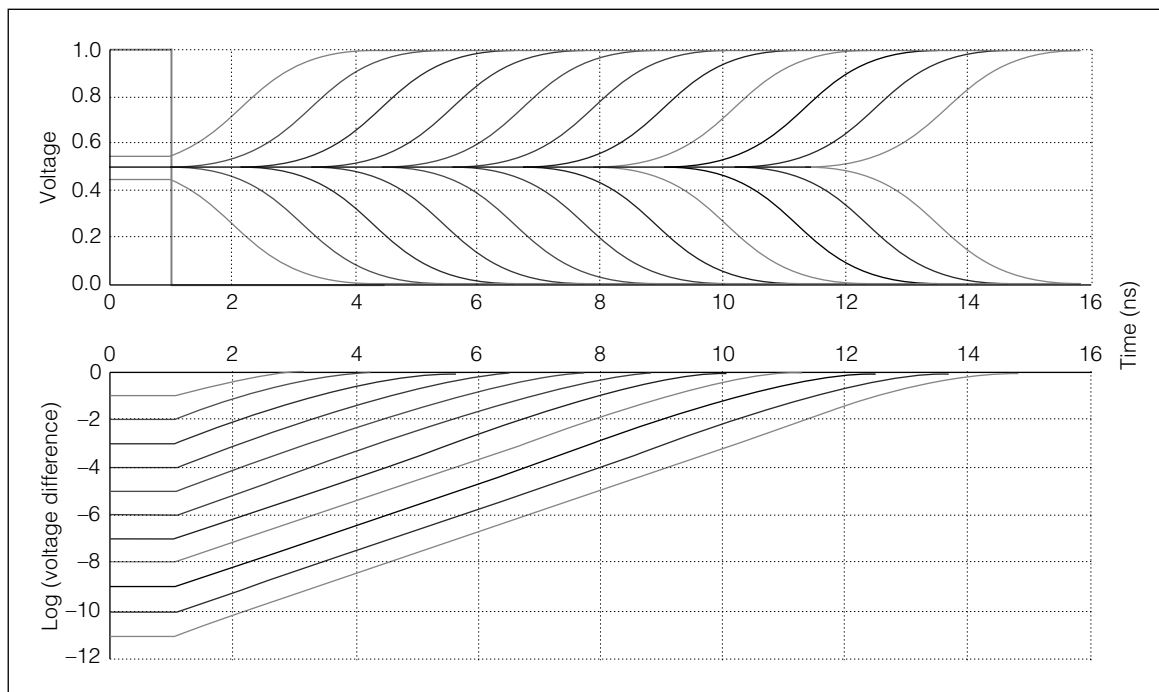


Figure 7. Simulations of metastability resolution with the starting voltage difference varying from 100 mV (left) to 10 pV (right); the lower the starting voltage, the longer resolution takes. The top chart shows voltage of the two latch nodes (the chart for $V_0 = 1 \mu\text{V}$ is the same as in Figure 5); the bottom chart shows the log of their difference (the line starting at -6 is the same as in Figure 6).

high or extremely low temperatures, could 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 :

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

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

Had we started with $V_0 = V_1$, then the time in metastability would be zero (the leftmost curve in Figure 7). On the other hand, if $V_0 = 0$, we would have waited forever, but this possibility is unlikely. In fact, the claim 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,” which some researchers and designers often make, should be taken lightly. Two factors affect the actual initial voltage: when exactly the clock’s sampling

edge blocked the input and closed the latch (specifically, what the actual value of V was at that moment), and how noise might have changed that value. Observe that thermal noise is anywhere from $1 \mu\text{V}$ to 1 mV , much higher than V in the extreme cases on the right-hand side of Figure 7. Since we don’t know V_0 deterministically, we don’t know how long the latch will stay metastable—but we can provide a statistical estimate. Probabilistic analysis shows that if a latch is metastable at time zero, the probability 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 would resolve pretty fast.

Synchronization reliability

All the foregoing leads us to computing reliability. If a latch receives asynchronous inputs, we can’t guarantee that it will never become metastable—in fact, we already know that it will definitely become metastable at the high rate of $F_D F_C T_W$ (2 million times/s in the preceding example). Instead, we can compute the reliability that it will fail as a result of entering metastability. The synchronizer’s whole purpose is to minimize that failure probability. Now we can finally

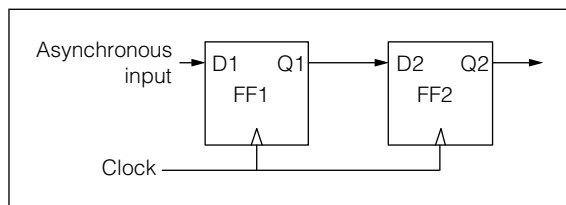


Figure 8. Two-flip-flop synchronization circuit.

define and estimate synchronization failures: we want 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 became metastable after the clock's sampling edge, and that it is still metastable S time later. The two events are independent, so we can multiply their probabilities:

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

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

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

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

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

Let's design synchronizers with MTBF that is many orders of magnitude longer than the expected lifetime of a given product. For instance, consider an ASIC designed for a 28-nm high-performance CMOS process. We estimate $\tau = 10$ ps, $T_W = 20$ ps (experimentally, we know that both parameters are close to the typical gate delay of the process technology), and $F_C = 1$ GHz. Let's assume that data changes every 10 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 we obtain 4×10^{29} years. (This figure should be 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, this is not exactly the case. 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.

Two-flip-flop synchronizer

Figure 8 shows a simple two-flip-flop synchronization circuit (we don't call it a synchronizer yet—that comes later). The first flip-flop (FF1) could become metastable. The second flip-flop (FF2) samples Q1 a cycle later; hence $S = T_C$. Actually, any logic and wire delays along the path from FF1 to FF2 are subtracted from the resolution time: $S = T_C - t_{pCQ}(\text{FF1}) - t_{\text{SETUP}}(\text{FF2}) - t_{pD}(\text{wire})$, and so on. A failure means that Q2 is unstable (it might change later than t_{pCQ}), and we know how to compute MTBF for that event. But what really happens inside the circuit? Consider Figure 9, in which D1 switches dangerously close to the rising clock. Any one of six outcomes could result:

- Q1 could switch at the beginning of clock cycle 1 and Q2 will copy that on clock cycle 2.
- Q1 could completely miss D1. It will surely rise on cycle 2, and Q2 will rise one cycle later.

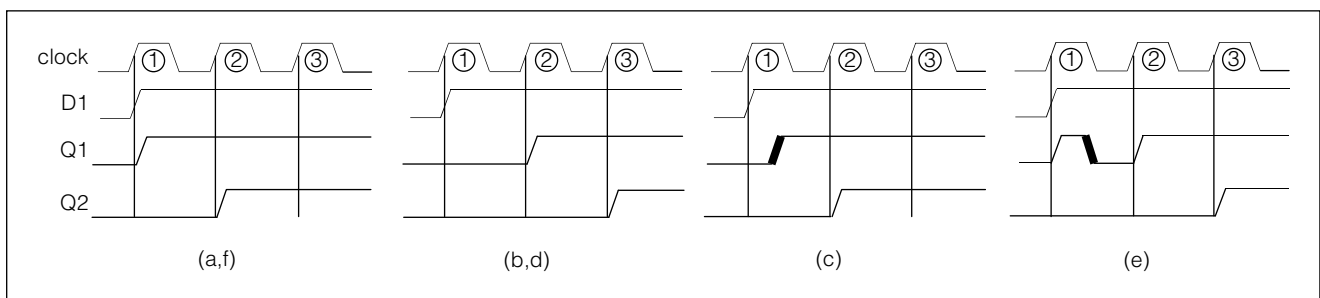


Figure 9. Alternative two-flip-flop synchronization waveforms.

- (c) FF1 could 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 could 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 case (a). Q2 rises in cycle 2.

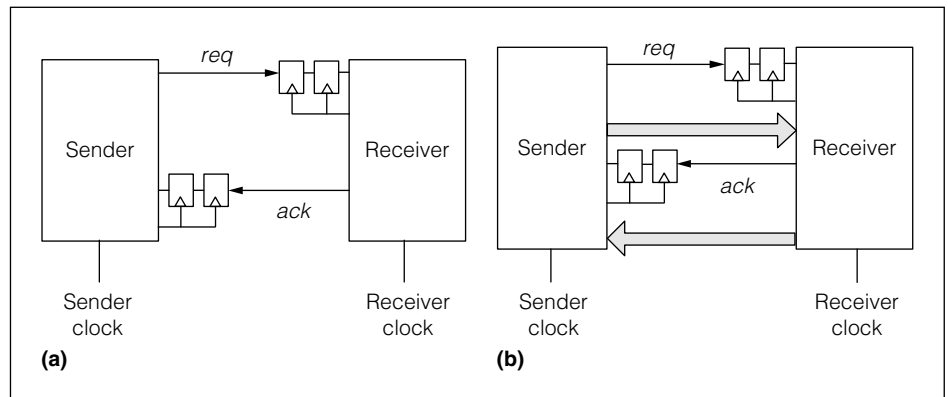


Figure 10. Complete two-way control synchronizer (a); complete two-way data synchronizer (b).

The bottom line is that 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 (continuous voltage levels changing over continuous time) for a simpler “digital” uncertainty (discrete voltage levels switching only at uncertain discrete time points) 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 might 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 continue reading to 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.

This, however, 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 can’t simply count cycles. To solve that, the receiver must send back an acknowledgment signal. Figure 10a shows a complete synchronizer. The sender sends *req* (also known as request, strobe, ready, or valid), *req* is synchronized by the top synchronization circuits, the receiver sends *ack* (or acknowledgment, or stall), *ack* is 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.

Now we can add data that needs to cross over, as in Figure 10b. 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 receives *ack*, it can store the received data and also start a new cycle.

Note that the synchronizer doesn’t 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 might pass through after one cycle, while others might take two cycles because of 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 might 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. As Figure 11a shows, when using slow clocks,

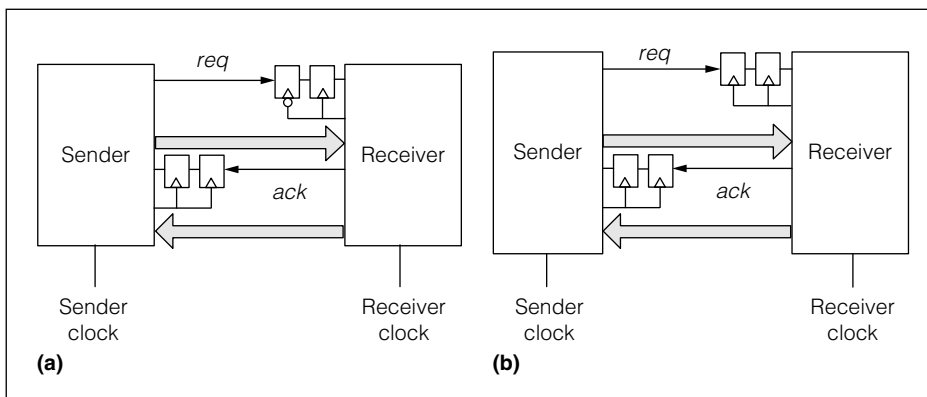


Figure 11. Variations on the theme of multi-flip-flop synchronization: half-cycle resolution for a very slow clock—the receiver in (a)—or three flip-flops enabling two cycles of resolution for the fast clocks and extreme conditions—the sender in (b). Because the sender and receiver could operate at widely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

resolution of less than half a cycle could suffice in the receiver side. In other cases, two flip-flops might not be enough. The clock may be fast (e.g., on processors that execute faster than 1 GHz), the supply voltage may go very low (especially in near-threshold designs), and the temperature may rise above 100 °C or drop far below freezing (for example, in a phone that will be used outdoors on a Scandinavian winter night or in a chip on the outside of an aircraft). For instance, if $S = T_C$, $F_C = 1$ GHz, $F_D = 1$ kHz (now we're being conservative), and due to low voltage and high temperature $\tau = 100$ ps and $T_W = 200$ ps, the MTBF is about one minute. Three flip-flops (the sender side in Figure 11b) would increase the MTBF a bit, to about one month. But if we use four flip-flops, $S = 3 T_C$ and the MTBF jumps to 1,000 years. Caution and careful design is the name of the game here.

Unique flip-flops designed especially for synchronization are more robust to variations in process, voltage, and temperature. Some use current sources to enhance the inverter gain; others sample multiple times and actively detect when synchronization is successful. The avid designer with the freedom to use nonstandard circuits can take advantage of such inventions, but typical ASIC and FPGA designers are usually constrained to using only standard flip-flops and will have to follow the usual, 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 uses 1,000 synchronizers, you

should be sure to design each one for MBTF at least three orders of magnitude higher than your reliability target for the entire system.

Similar concepts of synchronization are used for signals other than data that cross clock domains. Input signals might arrive at an 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 switches 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 *req*, two more cycles of the sender clock to receive *ack*, and possibly one more on each side to digest its input and change state. If *req* and *ack* must be lowered before new data can be transferred, consider another penalty of 3 + 3 cycles. (No wonder, then, that we used F_D much lower than F_C in the previous examples.) This slow pace is fine for many cases, but occasionally we want to work faster. Luckily, there are suitable solutions.

Two-clock FIFO synchronizer

The most common fast synchronizer uses a two-clock FIFO buffer as shown in Figure 12. Its advantages are hard to beat: you don't have to design it (it's typically available as a predesigned library element or IP core), and it's (usually) fast. The writer places data on the input bus and asserts *wen* (write enable); if *full* is not asserted, the data was accepted and stored. The reader asserts *ren* (read enable), and if *empty* is not asserted then data 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 respectively incremented.

When the read pointer points to the same word as the write pointer, the FIFO buffer 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* (read clock) when compared (on the right in Figure 12). That’s where the synchronization is; it’s 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 buffer, it might take one or two additional *rclk* cycles before the new write pointer passes through the synchronizer and deasserts *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 code so that only a single bit at a time changes in the synchronizer.

The FIFO solution usually works. It is nontrivial to design, but it’s often available in libraries and elsewhere. The key question for the user of a library FIFO buffer is how large the RAM should be (how deep should the buffer be). The common approach says “when in doubt, double it.” You might think that the life of the FPGA designer is less complex here—simply use trial and error. However, an FPGA has occasionally failed in mission because of a too-short FIFO buffer. Care is needed here.

The two-clock FIFO synchronizer, as well as a mixed-timed FIFO synchronizer interfacing a clock domain to an asynchronous (clockless) circuit, are used in a network on chip (NoC) that offers connectivity among many modules on a chip and also assumes full responsibility for synchronization. This is a refreshing idea: let someone else—the NoC vendor or designer—integrate your chip and take care of all clock domain crossings.

There are other fast synchronizers, which require a higher design effort than simply using the common FIFO synchronizer. The faster they are, the more complex the circuit. Most cases can be solved effectively with a good FIFO synchronizer. Two special cases are

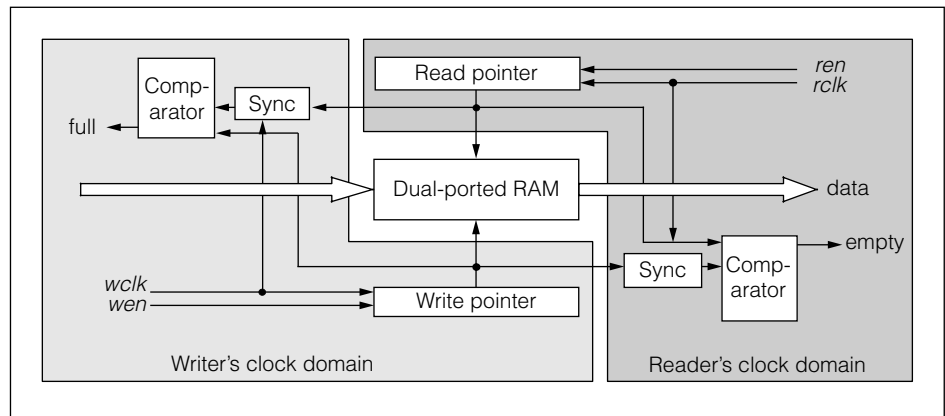


Figure 12. Two-clock FIFO synchronizer. It contains two separate clock domains and synchronizes pointers rather than data.

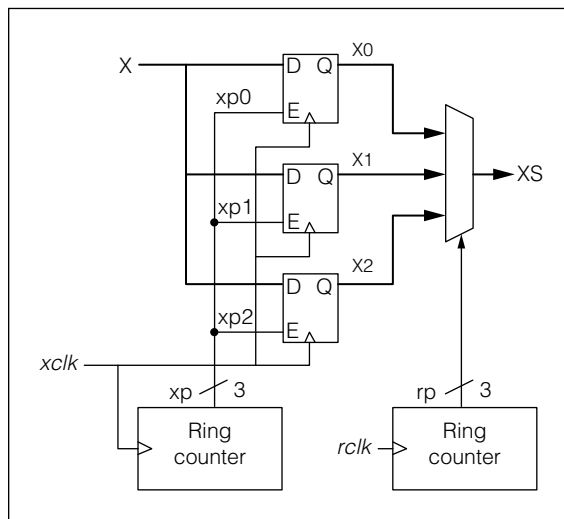


Figure 13. Mesochronous synchronizer. Read and write clocks share the same frequency but differ on phase. The reader ring counter selects the oldest available data word.

discussed next: one involves mesochronous clock domains; the other, synchronizing over long distance.

Mesochronous, multisynchronous, 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 might incur a heavy penalty in area and power). Once started, their relative phase remains stable. Figure 13 shows a common example: input X is

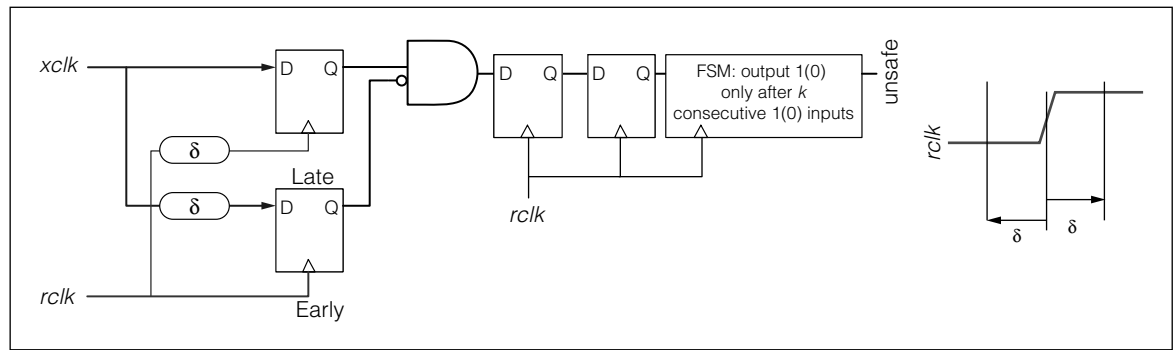


Figure 14. Conflict detector for multisynchronous synchronization.

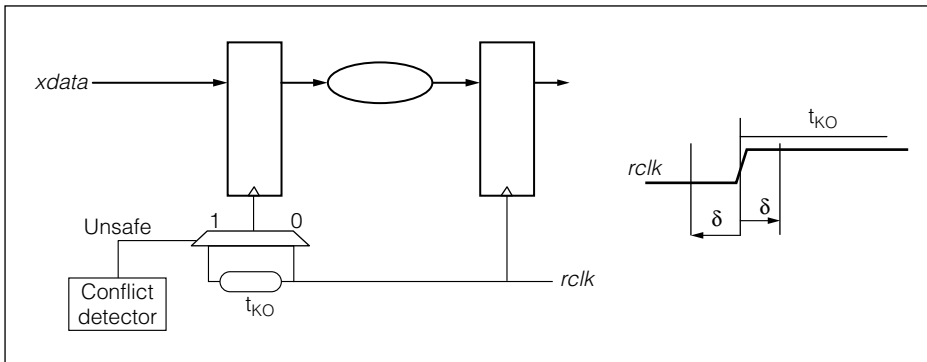


Figure 15. Synchronizer for multisynchronous domains.

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 previously discussed two-clock FIFO synchronizer (with at least four stages) can also do the job. It should incur a one- or two-cycle 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 synchronizer enables back pressure; when the receiver stops pulling data out, the sender is signaled *full* and can stall the data flow.

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 could lead to domains operating at the same frequency but at slowly changing relative phases. Such a relationship is termed *multisynchronous*, to distinguish this case from mesochronous operation. Synchronizers for multisynchronous domains need to continuously watch out for phase drifts and adapt to them. Figure 14

shows a conflict detector, which 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 (see 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 $rclk \pm \delta$. As before, the designer should also consider whether a simpler two-clock FIFO synchronizer could achieve the same purpose. Incidentally, in addition to on-chip clock domain crossings, multisynchronous domains exist in phase-adaptive SDRAM access circuits and in clock and data recovery circuits in high-speed serial link serializer/deserializer (SerDes) systems.

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 might get dangerously close to each other. But the conflict detector of Figure 14 is too slow to detect this on time (it could 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 might be suitable.

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

Different situations call for specific synchronizers that might excel given certain design parameters, but the conservative designer might well opt for the simpler, safer, commonly available two-clock FIFO synchronizer described earlier.

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 can't rely on counting cycles when waiting for the signal to arrive—process, voltage, and temperature variations as well as design variations (such as actual floor plan or placement and routing) might result in an unknown number of cycles for traversing the interconnecting wires.

The simplest (and slowest) approach is to stretch the simple synchronizers of Figure 10 over the distance. It's 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—for example, 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. Such a procedure is not trivial when *req* and the data wires are routed through completely different areas of the chip. This safety margin requirement usually results in even slower operation.

Using fast asynchronous channels somewhat mitigates the performance issue. Data bits are sent under the control of proper handshake protocols, and they're synchronized when reaching their destination. The downside is the need for special

IP cores, because asynchronous design is rarely practiced and isn't supported by common EDA tools.

Although the physical distance bounds the latency, throughput over long channels can be increased if we turn them into pipelines. But multistage pipelines require clocks at each stage, and it's not clear which clocks should be used in a multiclock-domain chip. When the data sent from Clock1 to Clock10 is 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 might lie in employing a NoC. The network infrastructure is intended to facilitate multiple transfers among multiple modules, over varying distances, to support 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.

Verification

Since the design of proper synchronization is such an elusive goal, verification is essential. But, for the same reason, verification is difficult and unfortunately doesn't always guarantee a correct solution.

Circuit simulations, such as that shown in 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 the two-flip-flop synchronizer discussion that a good synchronizer should contain all level and timing uncertainties, replacing them with 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, we can facilitate logic verification by replacing the synchronizer with a special synchronous delay block that inserts a delay of either k or $k + 1$ cycles at random. Although 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 is effective in detecting many logic errors (but not all of them—it wouldn't have helped the J spacecraft designers, for instance).

There are several EDA verification software tools, commonly called clock-domain-crossing (CDC) checkers, which identify and check all signals that

Literature Resources on Metastability

Synchronizers have been used as early as Eckert and Mauchly's ENIAC in the 1940s, but the first mathematical analysis of metastability and synchronizers was published in 1952,¹ and the first experimental reports of metastability appeared in 1973.² Other early works included those by Kinniment and Woods,³ Veendrick,⁴ Stucki and Cox,⁵ and Seitz.⁶ An early tutorial was published by Kleeman and Cantoni in 1987.⁷ Two books—by Meng⁸ and by Kinniment⁹—and two chapters in a book by Dally and Poulton¹⁰ were published on these topics. Several researchers have analyzed synchronizers,¹¹⁻¹³ and others have reported on metastability measurements.¹⁴⁻¹⁶ Beer et al. also observed that metastability parameters such as τ may deteriorate with future scaling, renewing concerns about proper synchronizer design.

Several researchers have described various methods for synchronizer simulation.^{11,12,17,18} Cummings detailed practical methods of creating two-clock FIFO synchronizers,¹⁹ and Chelcea and Nowick proposed mixed synchronous/asynchronous FIFO synchronizers—for example, for NoCs.²⁰ Dobkin et al.,²¹ as well as Dobkin and Ginosar,²² described fast synchronizers. Zhou et al.²³ and Kayam et al.²⁴ presented synchronizers that are robust to extreme conditions, improving on an earlier patent idea.²⁵ Ginosar and Kol discussed an early version of multisynchronous synchronization.²⁶ Ginosar described synchronizer errors and misperceptions.²⁷ Frank et al.²⁸ and Dobkin et al.²⁹ presented examples of formal synchronizer verification. Finally, Beigné et al. discussed employing a NoC to solve all synchronization issues.³⁰

References

1. S. Lubkin, "Asynchronous Signals in Digital Computers," *Mathematical Tables and Other Aids to Computation* (ACM section), vol. 6, no. 40, 1952, pp. 238-241.
2. T.J. Chaney and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Trans. Computers*, vol. C-22, no. 4, 1973, pp. 421-422.
3. D.J. Kinniment and J.V. Woods, "Synchronization and Arbitration Circuits in Digital Systems," *Proc. IEE*, vol. 123, no. 10, 1976, pp. 961-966.
4. H.J.M. Veendrick, "The Behavior of Flip-Flops Used as Synchronizers and Prediction of Their Failure Rate," *IEEE J. Solid-State Circuits*, vol. 15, no. 2, 1980, pp. 169-176.
5. M. Stucki and J. Cox, "Synchronization Strategies," *Proc. 1st Caltech Conf. VLSI*, Caltech, 1979, pp. 375-393.
6. C. Seitz, "System Timing," *Introduction to VLSI Systems*, chapter 7, C. Mean and L. Conway, eds., Addison-Wesley, 1979.
7. L. Kleeman and A. Cantoni, "Metastable Behavior in Digital Systems," *IEEE Design & Test*, vol. 4, no. 6, 1987, pp. 4-19.
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 Univ. Press, 1998.
11. C. Dike and E. Burton, "Miller and Noise Effects in a Synchronizing Flip-Flop," *IEEE J. Solid-State Circuits*, vol. 34, no. 6, 1999, pp. 849-855.

cross from one domain to another. Using a variety of structural design rules, they are helpful in assuring that no such crossing remains unchecked. Some assess overall SoC reliability in terms of MTBF. At least one tool also suggests solutions for problematic crossings in terms of synchronizer IP cores.

THIS SHORT TUTORIAL has been an attempt to present both the beauty and the criticality of the subject of metastability and synchronizers. For more than 65 years, many researchers and practitioners 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 an attractive, intriguing field. The folklore, the myths, the rumors, and the horror stories have added a fun aspect to a problem that has been blamed for the demise of several products and the large financial losses that resulted. Fortunately, with a clear understanding of the risks, respect for the dangers, and a strict engineering discipline, we can avoid the pitfalls and create safe, reliable, and profitable digital systems and products. ■

Acknowledgments

Chuck Seitz wrote the seminal chapter 7 of the VLSI "bible" (*Introduction to VLSI Systems*) and got me started on this. The late Charlie Molnar

12. D.J. Kinniment, A. Bystrov, and A. Yakovlev, "Synchronization Circuit Performance," *IEEE J. Solid-State Circuits*, vol. 37, no. 2, 2002, pp. 202-209.
13. S. Yang and M. Greenstreet, "Computing Synchronizer Failure Probabilities," *Proc. Design Automation and Test in Europe Conf. (DATE 07)*, EDAA, 2007, pp. 1-6.
14. L.-S. Kim, R. Cline, and R.W. Dutton, "Metastability of CMOS Latch/Flip-Flop," *Proc. IEEE Custom Integrated Circuits Conf. (CICC 89)*, IEEE Press, 1989, pp. 26.3/1-26.3/4.
15. Y. Semiat and R. Ginosar, "Timing Measurements of Synchronization Circuits," *Proc. IEEE Int'l Symp. Asynchronous Circuits and Systems (ASYNC 03)*, IEEE CS Press, 2003, pp. 68-77.
16. S. Beer et al., "The Devolution of Synchronizers," *Proc. IEEE Int'l Symp. Asynchronous Circuits and Systems (ASYNC 10)*, IEEE CS Press, 2010, pp. 94-103.
17. J. Jones, S. Yang, and M. Greenstreet, "Synchronizer Behavior and Analysis," *Proc. IEEE Int'l Symp. Circuits and Systems (ASYNC 09)*, IEEE CS Press, 2009, pp. 117-126.
18. S. Yang and M. Greenstreet, "Simulating Improbable Events," *Proc. 44th Design Automation Conf. (DAC 07)*, ACM Press, 2007, pp. 154-157.
19. C. 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 Trans. Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 8, 2004, pp. 857-873.
21. R. Dobkin, R. Ginosar, and C. Sotiriou, "High Rate Data Synchronization in GALS SoCs," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 10, 2006, pp. 1063-1074.
22. R. Dobkin and R. Ginosar, "Two Phase Synchronization with Sub-Cycle Latency," *Integration, the VLSI J.*, vol. 42, no. 3, 2009, pp. 367-375.
23. J. Zhou et al., "A Robust Synchronizer," *Proc. IEEE Symp. Emerging VLSI Technologies and Architectures (ISVLSI 06)*, IEEE CS Press, 2006, pp. 442-443.
24. M. Kayam, R. Ginosar, and C.E. Dike, "Symmetric Boost Synchronizer for Robust Low Voltage, Low Temperature Operation," EE tech. report, Technion, 2007; <http://webee.technion.ac.il/~ran/papers/KayamGinosarDike25Jan2007.pdf>.
25. R. Cline, *Method and Circuit for Improving Metastable Resolving Time in Low-Power Multi-State Devices*, US patent 5789945, to Philips Electronics North America Corporation, Patent and Trademark Office, 1998.
26. R. Ginosar and R. Kol, "Adaptive Synchronization," *Proc. Int'l Conf. Computer Design (ICCD 98)*, IEEE CS Press, 1998, pp. 188-189.
27. R. Ginosar, "Fourteen Ways to Fool Your Synchronizer," *Proc. IEEE Int'l Symp. Asynchronous Circuits and Systems (ASYNC 03)*, IEEE CS Press, 2003, pp. 89-96.
28. U. Frank, T. Kapschitz, and R. Ginosar, "A Predictive Synchronizer for Periodic Clock Domains," *J. Formal Methods in System Design*, vol. 28, no. 2, 2006, pp. 171-186.
29. R. Dobkin et al., "Assertion Based Verification of Multiple-Clock GALS Systems," *Proc. IFIP/IEEE Int'l Conf. Very Large Scale Integration (VLSI-SoC 08)*, 2008, pp. 152-155.
30. E. Beigné et al., "An Asynchronous NOC Architecture Providing Low Latency Service and Its Multi-Level Design Framework," *Proc. IEEE Int'l Symp. Asynchronous Circuits and Systems (ASYNC 05)*, IEEE CS Press, 2005, pp. 54-63.

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 of metastability and convinced me that this is a delightful subject.

Ran Ginosar is an associate professor of electrical engineering and computer science at the Technion—Israel Institute of Technology. His research interests include synchronization, asynchronous VLSI,

and VLSI architecture for parallel computing. He has a PhD in electrical engineering and computer science from Princeton University. He is a senior member of IEEE.

■ Direct questions and comments about this article to 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.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.