 By Michael Crews and Yong Yuenyongsgool, Philips Semiconductors

A SIMPLE CIRCUIT ADDRESSES THE ERRORS AND LIMITATIONS OF ASYNCHRONOUS DESIGN.

Practical design for transferring signals between clock domains

With the increasing integration of multiple systems on single SOCs (systems on chip) or boards, multiple clock frequencies in single digital designs have become common. Because of the asynchronous nature of these designs, passing data or control signals between logic operating on different clock frequencies presents a special set of problems. Because asynchronous design is unfamiliar to most experienced digital designers, errors are common. Many of these errors find their way into the silicon and even into production because they are nearly impossible to detect in simulation and easily missed in postsilicon validation. Problems of performance degradation, back-end EDA-tool incompatibility, and dependency on the frequency relationship of the clocks involved often plague even functionally correct implementations. Frequency dependency is problematic for production tests that run the parts at different speeds, and it limits reusability in future designs with different system frequencies. You can address all of these errors and limitations with a fairly simple circuit that works for both data and control logic.

The circuit requires both signal synchronization and a handshake protocol (**Figure 1**). The synchronizer guarantees the amount of time required for the signal level to settle following a metastability violation, thereby preventing undetermined signal levels from propagating to the destination module. The handshake protocol maintains signals levels long enough to ensure that the system does not miss signal events or wrongly interpret them as multiple events. Normally, the circuit synchronizes only handshake signals, which signify the validity of data being transferred to the destination clock domain. Once the handshake signals transfer to the destination-clock domain, the system clocks the data set directly to the destination module. The most common mistakes in this situation involve the handshake system and its usage.

SYNCHRONIZATION

A signal that a system sends from one clock domain arrives as an asynchronous signal in the destination-clock domain, possibly violating the destination flip-flop setup or hold time, causing it to enter a metastable condition. This condition, in turn, causes propagation of nonbinary signals to other parts of the system. The time required for the metastable flip-flop to settle out to a binary voltage level varies. A double-stage synchronizer (**Figure 2**) is the most widely used method of stabilizing a signal in the destination-clock domain. If the first flip-flop stage enters the metastable condition, it has a full clock period to stabilize before the second flip-flop stage samples it. Only the second-stage value is propagated to other parts of the system. To ensure that interconnect delay does not reduce the one-clock-period settling time that the double-stage synchronizer supplies, you must minimize interconnect delay between the two stages. To do so, you place the flop-flops directly next to each other on the die or board. Note that you can add synchronization stages in series to reduce the probability of the metastable condition's propagating to the last stage, but you pay a price in system performance. However, the dou-

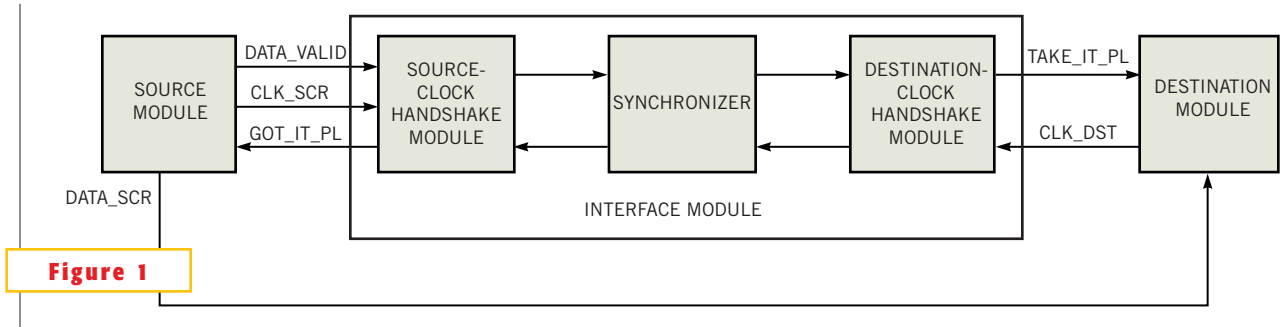


Figure 1

A simple circuit employing a synchronizer and handshake protocol can help overcome limitations and errors inherent in asynchronous design.

ble-stage synchronizer is generally sufficient, even for high-speed systems, because metastable settling time scales down with the technology that enables high-speed designs.

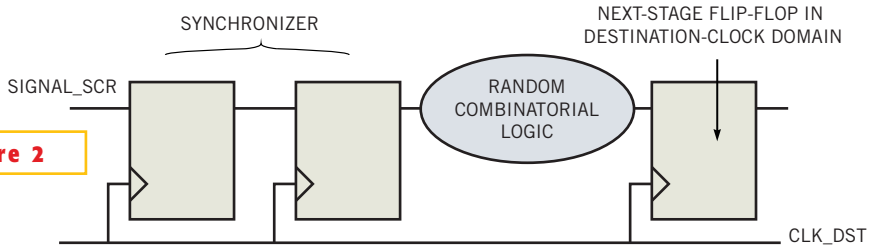


Figure 2

A double-stage synchronizer is the most widely used method of stabilizing a signal in the destination-clock domain.

HANDSHAKE PROTOCOL

A double-stage synchronizer stabilizes a signal in the destination-clock domain, but it does not ensure that the signal remains stable long enough for the destination circuit to sample it once and only once. Consider a case in which a system asserts a signal in a fast source-clock domain and then sends it to a very slow destination-clock domain. The source must hold the signal for multiple clocks, or the logic of the slower destination-clock domain may never detect it. On the other hand, if the system asserts a signal in a slow source-clock domain and then sends it to a very fast destination-clock domain, the logic may detect the signal multiple times, mistaking it for multiple events. A handshake protocol can solve this problem.

Figure 1 shows an interface between two modules operating in different clock domains. The interface module contains the synchronizers and the handshake-protocol logic. The handshake protocol must ensure that the data holds stable long enough for circuitry to sample it in the destination-clock domain. It must also ensure that the handshake-protocol logic does not signal a new data-valid signal until the destination has acknowledged the first data valid signal was received. Failing to recognize the need for this data-valid deassertion acknowledgment is a common failure of asynchronous designs.

One of the most efficient protocol implementations is the "toggle" implementation. By using the change in the handshake's signal level and not the level itself to communicate through the synchronizer, the system immediately readies itself for another transaction. The deassertion acknowledgment occurs without the need for a second round trip to restore all control signals to their proper logical states, as a four-phase round trip requires. **Figure 3** illustrates this protocol. The figure shows no clocks, because the protocol works with any clock-frequency relationship between source and destination.

When the source module initiates a write transaction to the destination module, the interface module responds by asserting the START_PL signal. This signal begins the transfer process. Sampling the acknowledgment signal GOT_IT_PL deasserted, the source module continues driving the DATA_SRC and DATA_VALID signals, therefore allowing the time for the destination logic to capture the DATA_SRC in the destination-clock domain. The TAKE_IT_TG signal, derived from the START_PL signal, is the toggle-handshake signal. Once clocked through the synchronizer, its pulse derivative, TAKE_IT_PL, is generated in the destination-clock domain. The TAKE_IT_PL signal guarantees the stability of the DATA_SRC data signal and acts as a strobe to capture the DATA_SRC signal in the destination-clock domain.

By the time the toggle handshake signal has been clocked through the synchronizer and appears in the destination-clock domain, the associated set of data, which the source model has been continuously driving, is stable with respect to the destination-clock domain. Therefore, it is safe to clock the data into the destination-clock domain without synchronizing the data signal. The TAKE_IT_PL signal generates the toggle-handshake counterpart, whereupon it is clocked through the other synchronizer back to the source-module clock domain. As a result, the GOT_IT_PL signal is generated in the source-module clock domain and signifies the completion of the current transaction, allowing the source module to resume its normal operation. Notice that the system does not restore the logical levels of the toggle-handshake signals, TAKE_IT_TG and GOT_IT_TG, to their original states. Again, the transitional—not the logical—level acts as a means of communication across clock domains. **Figure 4** illustrates this situation. Note that, because the handshake protocol ensures stable data, the data signals need not be synchronized themselves.

You can also adapt this method to sup-

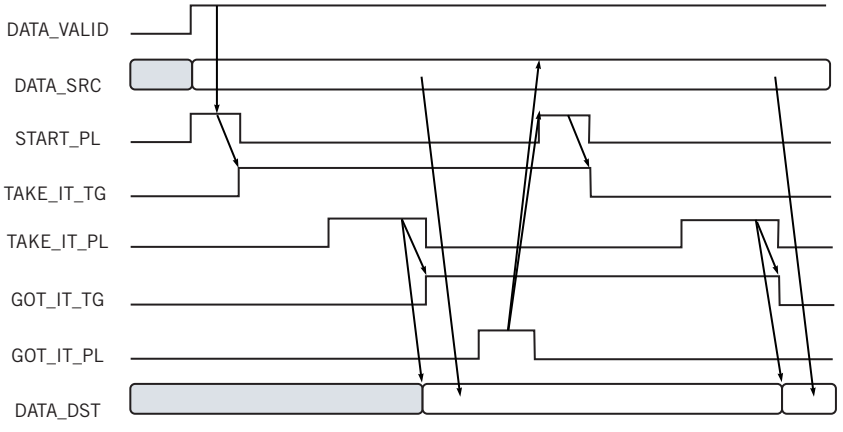
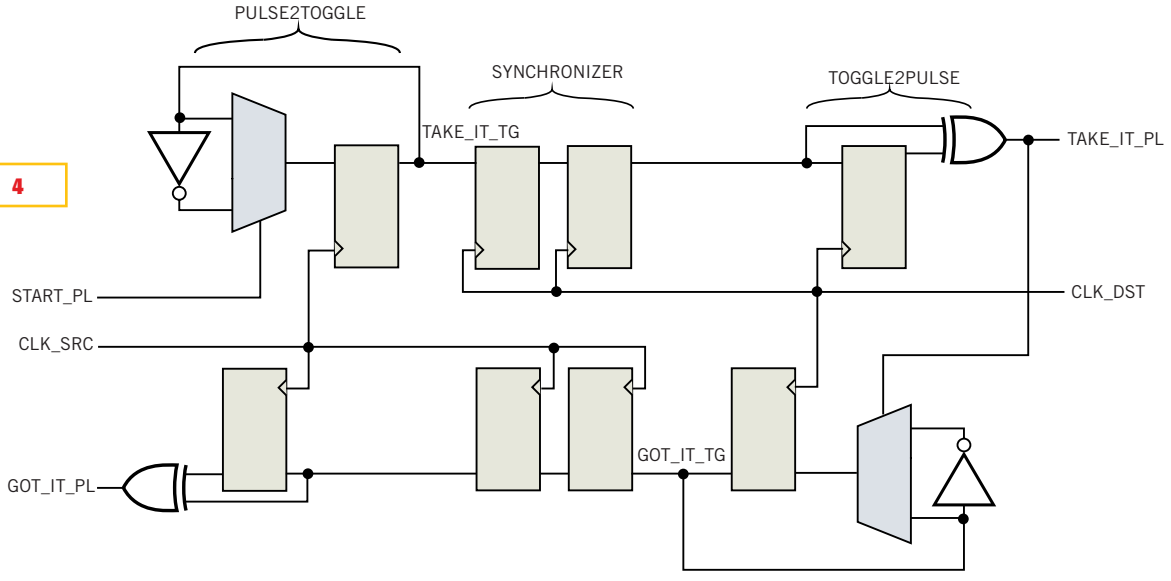


Figure 3

The toggle-handshake protocol uses the change of the handshake's signal level—not the level itself—to communicate through the synchronizer, and the system immediately readies itself for another transaction.

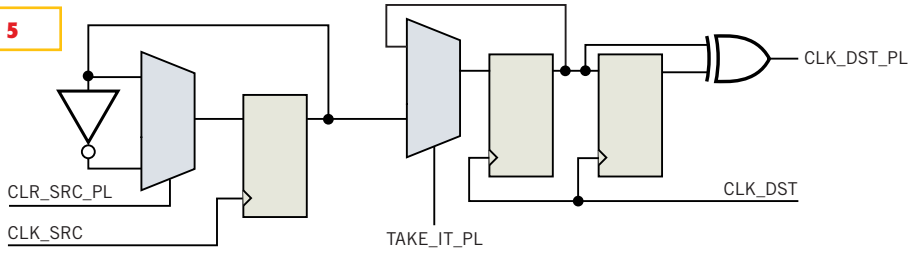
Figure 4



In the basic double-stage synchronization-and-toggle protocol, the transitional–not the logical–level acts as a means of communication across clock domains.

port self-clearing bits, such as those common in interrupt-set and -clear registers. In such cases, the CPU's service routine must clear the interrupt by writing a self-clearing bit in the destination module's register. It achieves this feat through the use of pulse-to-toggle and toggle-to-pulse converters (**Figure 5**). Note that a self-clearing bit is logically equivalent to a pulse.

Figure 5



Modifying the system in Figure 4 allows it to support a self-clearing register bit.

Due to the difference in clock speeds between modules, a latency-absorbing FIFO often acts as a data buffer destined for a different clock domain. In this case, the FIFO empty and full conditions perform the handshake (**Figure 6**). This situation requires the FIFO to pass the input and output pointers between clock domains. Because the pointers contain multiple bits, they can introduce a race condition through the synchronizer. To avoid this problem, you must implement the input and output pointers as Gray Code counters to ensure that only one bit changes at a time.

AVOID THE PITFALLS

One of the most common pitfalls of asynchronous design involves the use of level-handshake protocols. Similar to the toggle method, a strobe from the source clock domain generates a level handshake signal, which gets clocked through a synchronizer in the destination-clock domain. The signal's level, as opposed to its transition, is a means of communication. The level-handshake signal in the destination-clock domain is then clocked through another synchronizer to acknowledge that the source-clock domain has completed the transfer. This transaction may seem complete; however, it does not restore the handshake signals to their original values in preparation for the next handshake.

To resolve this situation, you must initiate another level round trip to restore the logical level of the handshake signal to its original state—doubling the protocol latency. Employing asynchronous flip-flops as a means of restoring the handshake signal level to its original state in the one round-trip level protocol seems a natural way to cope with this issue. Though you can work out the method to yield the correct logical functions, using asynchronous flip-flops significantly complicates static-timing analysis and manufacturing test.

Another common pitfall includes clocking of the data, as well as the strobe signal, through the synchronizers, thus causing a race condition. The race condition in this situation occurs when the data and strobe signals enter a metastable state at slightly different times, environmental conditions, or both. In this situation, it is impossible to guarantee the resolved logical states of both strobe signal and data.

Sensible solutions must neither assume nor require any fixed relationship between the source- and destination-clock frequencies. An implementation with a fixed-frequency relationship significantly limits the reusability of the design and imposes significant limitation for manufacturing test. With any design implementation, verification is critical. However, simulation cannot hope to duplicate the infinite number of clock and signal-edge relationships that are possible in a clock-domain-crossing design. Therefore, the only complete form of verification is inspection. When inspecting a clock-domain-crossing design, you must fully analyze two cases. First, assume one extreme clock-frequency relationship (10 to one, for example) and manually analyze the behavior of the implementation on a timing diagram for at least two consecutive transactions. Then,

repeat the analysis, assuming the opposite of the clock-frequency relationship. Of course, the best way to avoid errors is to stick with a standard implementation, such as the toggle technique.□

AUTHORS' BIOGRAPHIES

Michael Crews is a design-engineering manager at Philips Semiconductors, where he has worked for seven years. He leads a hardware-design group that develops multimillion-gate SOCs for media-processing markets. He holds a BSEE from Arizona State University (Tempe) and enjoys piloting planes, scuba diving, and traveling.

Yong Yuenyongsgool is a design engineer at Philips Semiconductors, where he has worked for seven years. He is responsible for the digital design and verification of embedded multimedia processor chips. He holds a BSEE and an MSEE from Arizona State University (Tempe) and enjoys jogging, hiking, and community service.

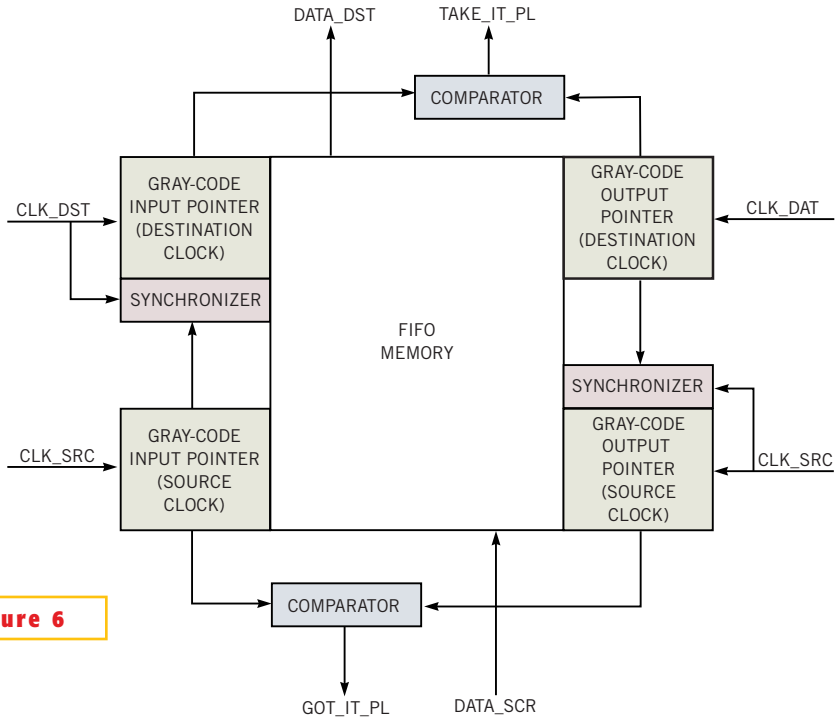


Figure 6

Due to the difference in clock speeds between modules, a latency-absorbing FIFO often acts as a buffer data destined for a different clock domain.