 By Mike Stein, Paradigm Works

AS DIGITAL DESIGN BECOMES INCREASINGLY SOPHISTICATED, CIRCUITS WITH MULTIPLE CLOCKS MUST RELIABLY COMMUNICATE WITH EACH OTHER.

Crossing the abyss: asynchronous signals in a synchronous world

Only the most elementary logic circuits use a single clock. Most data-movement applications, including disk-drive controllers, CD-ROM/DVD controllers, modems, network interfaces, and network processors, bear inherent challenges moving data across multiple clock domains. When signals travel from one clock domain to another, the signal appears to be asynchronous in the new clock domain.

In modern IC, ASIC, and FPGA designs, engineers have many software programs to help them create million-gate circuits, but these programs do not solve the problem of signal synchronization. It is up to the designer to know reliable design techniques that reduce the risk of failure for circuits communicating across clock domains.

FUNDAMENTALS

The first step in managing multiclock designs is to understand the problem of signal stability. When a signal crosses a clock domain, it appears to the circuitry in the new clock domain as an asynchronous signal. The circuit that receives this signal needs to synchronize it. Synchronization prevents the metastable state of the first storage element (flip-flop) in the new clock domain from propagating through the circuit.

Metastability is the inability of a flip-flop to arrive at a known state in a specific amount of time. When a flip-flop enters a metastable state, you can predict neither the element's output voltage level nor when the output will settle to a correct voltage level. During this settling time, the flip-flop's output is at some intermediate voltage level or may oscillate and can cascade the invalid output level to flip-flops farther down the signal path.

The input must be stable during a small window of time around the active edge of the clock for any flip-flop. This window of time is a function of the design of the flip-flop, the implementation technology, operating conditions, and the load on the output for outputs that are not buffered. Sharp edge rates on the input signal minimize the window. More windows of vulnerability arise as the clock frequency increases, and the probability of hitting the window increases as the data frequency increases.

FPGA manufacturers and IC foundries qualify their flip-flops and determine their characteristics. "MTBF" (mean time between failures) describes the metastability characteristic of a flip-flop using statistics to determine the probability of a flip-flop's failure. Manufacturers base the MTBF in part on the length of the time window during which a change in the input signal causes the flip-flop to become unstable. In addition, MTBF calculation uses the frequency of the input signal and the frequency of the clock driving the flip-flop.

Each type of flip-flop in an ASIC or FPGA library has timing requirements to help you determine the window of vulnerability. "Setup time" describes the time an input signal to a flip-flop must be stable before the clock edge. "Hold time" is the time the signal must remain stable after the clock edge. These specifications are usually conservative to account for all the possible variations in supply voltage, operat-

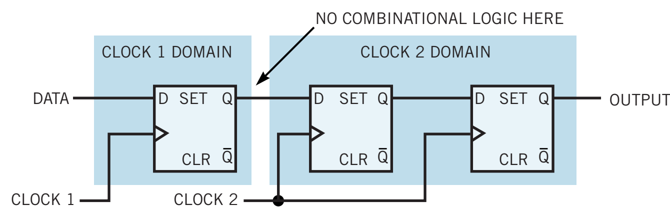
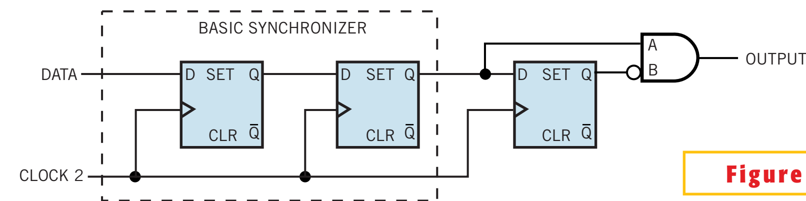


Figure 1 In a full synchronizer circuit, the signal-crossing clock domains should pass from the originating flip-flop in the original clock domain to the first flip-flop of the synchronizer without passing through any combinational logic between the originating flip-flop and the first flip-flop of the synchronizer.

ing temperature, signal quality, and fabrication. If a design meets these timing requirements, the possibility is negligible that the flip-flop will fail.

Synthesis programs in modern IC and FPGA designs ensure that digital circuits meet the setup-and-hold requirements for each flip-flop in the design; however, asynchronous signals pose problems for the software. A signal crossing a clock domain appears to be asynchronous to the logic in the new clock domain. Most synthesis programs have trouble determining whether asynchronous signals meet the timing requirements for flip-flops. Because they cannot determine the time the flip-flop is unstable, they cannot determine the total delay from the flip-flop through the combinational logic to the next flip-flop. The best course, then, is to use circuits that mitigate the impact of asynchronous signaling.



The edge-detecting synchronizer circuit adds a flip-flop to the output of the level synchronizer.

SIGNAL SYNCHRONIZATION

The purpose of synchronizing signals is to protect downstream logic from the metastable state of the first flip-flop in a new clock domain. A simple synchronizer comprises two flip-flops in series without any combinational circuitry between them. This design ensures that the first flip-flop exits its metastable state and its output settles before the second flip-flop samples it. You also need to place the flip-flops close to each other to ensure the smallest possible clock skew between them.

IC foundries help with signal synchronization by providing synchronizer cells. These cells usually comprise a flip-flop with a very high gain that uses more power and is larger than a standard flip-flop. Such a flip-flop has reduced setup-and-hold-time requirements for the input signal and is resistant to oscillation when the input signal causes a metastable condition. Another type of synchronizer cell contains two flip-flops, thus easing your job by placing the flip-flops close to each other and preventing you from placing any combinational logic between them.

For synchronization to work properly, the signal crossing a clock domain should pass from flip-flop in the original clock domain to the first flip-flop of the synchronizer without passing through any combinational logic between the two (**Figure 1**). This requirement is important because the first stage of a synchronizer is sensitive to glitches that combination logic produces. A long enough glitch that occurs at the correct time could meet the setup-and-hold requirements of the first flip-flop in the synchronizer, leading the synchronizer to pass a false-valid indication to the rest of the logic in the new clock domain.

A synchronized signal is valid in the new clock domain after two clock edges. The signal delay is between one and two clock periods in the new clock domain. A rule of thumb is that a synchronizer circuit causes two clock cycles of delay in the new clock domain, and a designer needs to consider how synchronization delay impacts timing of signals crossing clock domains.

There are many designs for synchronizers because one type does not work well in all applications. Synchronizers fall into one of three basic categories: level, edge-detecting, and pulse (**Table 1**). Other synchronizer designs exist, but these serve for most applications a designer encounters. In a level synchronizer, the signal crossing a clock domain stays high and stays low for more than two clock cycles in the new clock domain. A requirement of this circuit is that the signal needs to change to its invalid state before it can become valid again. Each time the signal goes valid, the receiving logic considers it a single event, no matter how long the signal remains valid. This circuit is the heart of all other synchronizers.

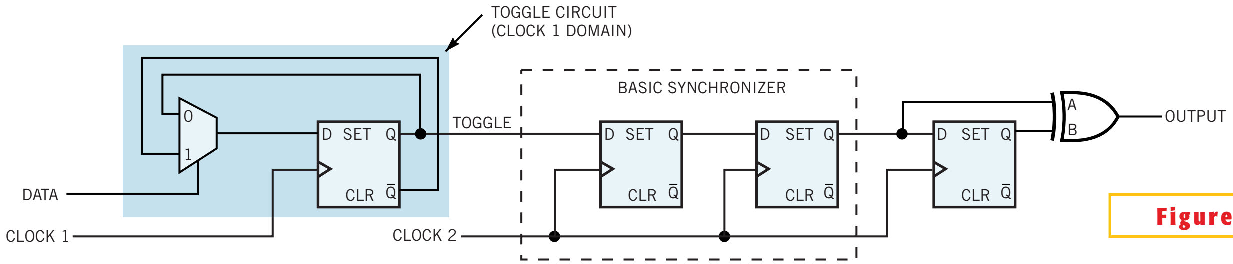
The edge-detecting synchronizer circuit adds a flip-flop to the output of the level synchronizer (**Figure 2**). The output of the additional flip-flop is inverted and ANDed with the output of the level synchronizer. This circuit detects the rising edge of the input to the synchronizer and generates a clockwide, active-high pulse. Switching the inverter on the AND gate inputs creates a synchronizer that detects the falling edge of the input signal. Changing the AND gate to a NAND gate results in a circuit that generates an active-low pulse.

The edge-detecting synchronizer works well at synchronizing a pulse going to a faster clock domain. This circuit produces a pulse that indicates the rising or falling edge of the input signal. One restriction of this synchronizer is that the width of the input pulse must be greater than the period of the synchronizer clock plus the required hold time of the first synchronizer flip-flop. The safest pulse width is twice the synchronizer clock period. This synchronizer does not work if the input is a single clockwide pulse entering a slower clock domain; however, the pulse synchronizer solves this problem.

The input signal of a pulse synchronizer is a single clockwide pulse that triggers a toggle circuit in the originating clock domain (**Figure 3**). The output of the toggle circuit switches from high to low and vice versa each time it receives a pulse and passes through the level synchronizer to arrive at one input of the XOR gate, while a one-clock-cycle-delayed version goes to the other input of the XOR. For one clock cycle, each time the toggle circuit changes state, the output of this synchronizer generates a single clockwide pulse.

TABLE 1–SYNCHRONIZER TYPES AND USES

Type	Application	Input	Output	Restriction
Level signals	Synchronizes level	Level	Level	Input must be valid for at least two clock periods in the new domain. Each time output goes valid counts as a single event.
Edge-detecting	Detects rising or falling edge of input	Level or pulse	Pulse	Input must be valid for at least two clock periods in the new domain.
Pulse	Synchronizes single clockwide pulses	Pulse	Pulse	Input pulses must have at least two clock periods between them in the new domain.



The input signal of a pulse synchronizer is a single-clock, cyclewide pulse that triggers a toggle circuit in the originating clock domain.

The basic function of a pulse synchronizer is to take a single clockwide pulse from one clock domain and create a single clockwide pulse in the new domain. One restriction of a pulse synchronizer is that input pulses must have a minimum spacing between pulses equal to two synchronizer clock periods. If the input pulses are closer, the output pulses in the new clock domain are adjacent to each other, resulting in an output pulse that is wider than one clock cycle. This problem is more severe when the clock period of input pulse is greater than twice the synchronizer clock period. In this case, if the input pulses are too close, the synchronizer does not detect every one.

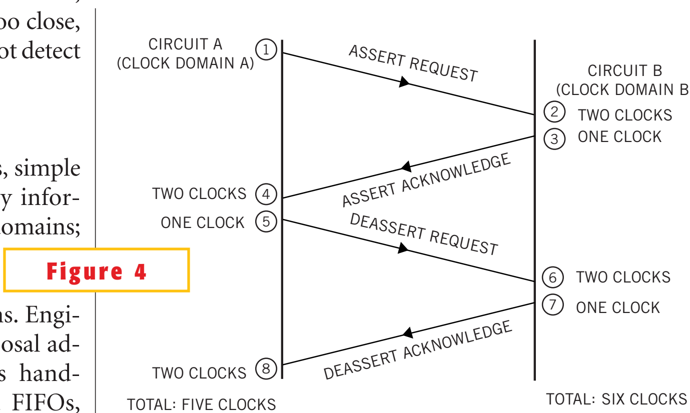
HANDSHAKING AND FIFOs

In many applications, simple signals are not the only information crossing clock domains; data, address, and control buses also travel together across domains. Engineers have at their disposal additional tools, such as handshaking protocols and FIFOs, that can handle these situations.

Handshaking allows digital circuits to effectively communicate with each other when the response time of one or both circuits is unpredictable. For example, an arbitrated bus allows more than one circuit to request access to a single bus, such as PCI or AMBA (Advanced Microcontroller Bus Architecture), using arbitration to determine which circuit gains access to the bus. Each circuit signals a request, and the arbitration logic determines which request "wins." This winning circuit receives an acknowledgment indicating that it has access to the bus. It then discontinues its request and begins the bus transaction.

Full- and partial-handshake signaling are the two fundamental types of handshake protocol that circuits on different clock domains use. Each type of handshake uses synchronizers, and each has its own set of design trade-offs. In full-handshake signaling, the two circuits wait for each other before asserting or dropping their respective handshake signal (**Figure 4**). First, Circuit A asserts its request signal. Next, Circuit B detects that the request signal is valid and asserts its acknowledgment signal. When Circuit A detects that the acknowledgment signal is valid, it drops its request signal. Finally, when Circuit B detects that the request is invalid, it drops its acknowledgment signal. Circuit A does not make a new request until it detects that the acknowledgment signal is invalid.

This type of handshake uses level synchronizers. A designer uses this technique when the acknowledging circuit (Circuit B) needs to inform the requesting circuit (Circuit A) that it is actively processing the request. This handshake requires that the requesting circuit hold off its next request until it detects that the ac-

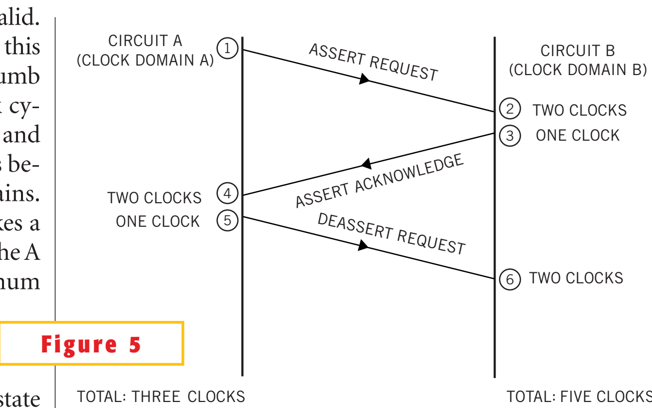


In full-handshake signaling, the two circuits wait for each other before asserting or dropping their respective handshake signals.

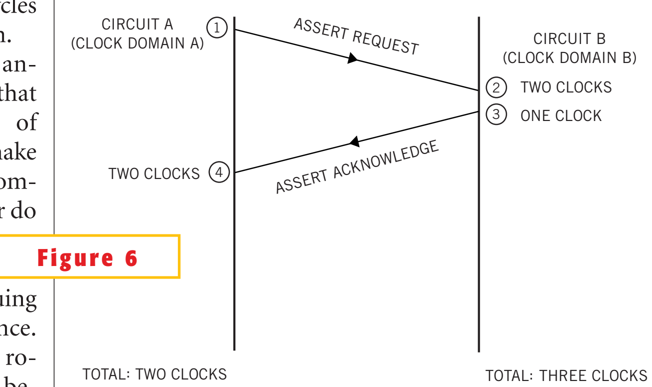
TABLE 2–HANDSHAKING TECHNIQUES

Handshake type	Circuits	Signaling type	Sequence length	Synchronizer	Restrictions
Full	Circuit A (request)	Level	Five clocks	Level	Sequence is long. Request must be invalid for at least two of the Circuit B clock periods.
	Circuit B (acknowledge)	Level	Six clocks	Level	Acknowledgment must be invalid for at least two of the Circuit A clock periods.
Partial I	Circuit A (request)	Level	Three clocks	Pulse or edge-detect	Must control rate of acknowledgment pulses.
	Circuit B (acknowledge)	Pulse	Five clocks	Level	Request must be invalid for at least two of the Circuit B clock periods.
Partial II	Circuit A (request)	Pulse	Two clocks	Pulse or edge-detect	Must save pending request information.
	Circuit B (acknowledge)	Pulse	Three clocks	Pulse or edge-detect	Must register request and acknowledgment signals.

knowledgment signal is invalid. To determine the timing for this protocol, use the rules of thumb that signals take two clock cycles to cross a clock domain and that circuits register signals before they cross clock domains. The complete sequence takes a maximum of five cycles in the A clock domain plus a maximum of six cycles in the B clock domain. Full handshaking is robust because each circuit explicitly knows the state of the other by examining the request and acknowledgment signals. The drawback of this scheme is that the entire process uses many clock cycles to complete the transaction.



In a partial-handshake scheme, Circuit A asserts its request signal, and Circuit B acknowledges it with a single clockwide pulse.



This type of partial-handshake scheme uses pulse synchronizers, but a circuit that has a clock that is twice as fast as the other can instead use an edge-detecting synchronizer.

Partial handshaking is another signaling technique that shortens this sequence of events. With partial handshake signaling, the two circuits communicating with each other do not wait for the other one before dropping their respective signal and continuing with the handshake sequence. Partial handshaking is less robust than full handshaking because the handshake signals do not indicate the state of both circuits; each circuit must save state information normally present in full handshake signals. However, by not waiting until the other circuit drops it handshake signal, the whole sequence of events takes less time.

When using partial-handshake signaling, the acknowledging circuit must generate its signal at the correct time. If the acknowledging circuit needs to complete processing the request before it can handle another, then the timing of the acknowledgment signal is important. The circuit uses its acknowledgment signal to indicate when it completed any processing. One partial-handshake scheme mixes level and pulse signaling, and the other uses only pulse signaling.

In the first partial-handshake scheme, Circuit A asserts its request signal as an active level, and Circuit B acknowledges it with a single clockwide pulse. In this case, Circuit B does not care when Circuit A drops its request signal. However, to make this technique work, Circuit A must drop its request signal for at least one clock cycle; otherwise, Circuit B cannot distinguish between a previous request and a new request. With this handshake, Circuit B uses a level synchronizer for the request signal, and Circuit A uses a pulse synchronizer for the acknowledgment signal. The acknowledgment pulses occur only when Circuit B detects the request signal. This situation allows Circuit A to control the spacing of pulses it receives into the synchronizer by controlling the timing of its request signal (**Figure 5**). Once again, to determine timing, use the rules of thumb that signals take two clock cycles to cross a clock domain and that circuits register signals before they cross clock domains.

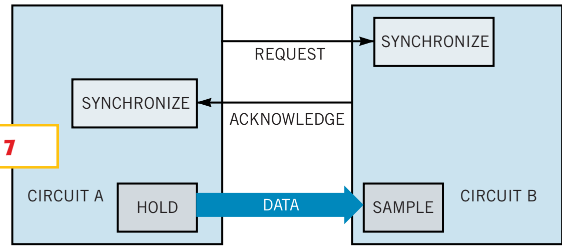
The complete sequence takes a maximum of three cycles in the A clock domain plus a maximum of five cycles in the B clock domain. This partial handshake signaling uses two fewer clock cycles in the A clock domain and one fewer clock cycle in the B clock domain than full handshake signaling. You can shave off a few more clock cycles using a second partial-handshake scheme where Circuit A asserts its request with a single clockwide pulse and Circuit B acknowledges the request with a single clockwide pulse. In this case, both circuits need to save state to indicate that the request is pending.

This type of handshake uses pulse synchronizers, but either circuit that has a clock that is twice as fast as the other can instead use an edge-detecting synchronizer instead (**Figure 6**). The complete sequence takes a maximum of two cycles in the A clock domain plus a maximum of three cycles in the B clock domain. This partial-handshaking technique uses three fewer clock cycles in the A clock domain and three fewer clock cycle in the B clock domain than full-handshake signaling. This technique is also faster than the first partial-handshake signaling by one cycle in the A clock domain and two cycles in the B clock domain (**Table 2**). These handshake protocols involve single signals that cross clock domains. However, when groups of signals cross clock domains, designers need to use more complex signaling schemes.

DATAPATH DESIGN

One important rule when synchronizing signals is that a design should not synchronize the same signal in more than one place; that is, a single signal fans out to more than one synchronizer. Because synchronization takes one to two clock cycles, a designer cannot reliably predict when each of these signals arrives across a clock domain. In addition, the timing of a group of synchronized signals in the new clock domain can vary because the delay can be either a single clock cycle of delay or two cycles, depending on when the input arrives at the synchronizer. This situation results in a "race condition" between the individually synchronized signals. This race condition also applies to groups of signals, such as data, address, and control buses, that need to travel together across clock domains. Thus, you should not use individual synchronizers

on each signal in the group or on each bit of a data bus or an address bus in which each signal needs to be valid in the new clock domains at the same time.



A datapath-synchronizer design uses a holding register and handshake signaling.

One way to solve the problem of bus synchronization is to use a holding register and handshake signaling. This circuit comprises a register that holds the signal bus and a handshaking scheme (**Figure 7**). The handshake signals indicate when the circuit in the new clock domain can sample the bus and when the originating circuit can replace the current contents of the holding register.

In this design, the transmitting circuit stores the data (signal bus) in the holding register as it asserts the request signal. These two actions can happen at once because the request signal takes at least one clock cycle before the receiving circuit detects it (the minimum handshake-synchronization delay). When the receiving circuit samples the data (signal bus), it asserts the acknowledgment signal. This design uses full handshaking and takes a long time to complete the transfer. A design using full-handshake signaling has a large window of time for the receiving circuit to sample the signal bus and is thus somewhat inefficient. The same design can use a partial handshake instead of a full handshake to speed the transfer.

With this type of bus synchronization, you synchronize the handshake signals but not the signal bus. The signal bus originates from the holding register and remains stable until after the receiving circuit samples it. Note that bus synchronization may not work in applications in which the transmitting circuit presents data too fast for the receiving circuit to handle it.

ADVANCED DATAPATH DESIGN

In many cases, data needs to "pile up" as it crosses clock domains, so designs using a single holding register do not work. One such case is a transmitting circuit that presents data in bursts, too quickly for a receiving circuit to sample. Another such case is a receiving circuit that samples data faster, but in a narrower data width, than the transmitting circuit sends it. These situations call for the use of a FIFO.

Fundamentally, a designer uses a FIFO for speed matching, data-width matching, or both. For speed matching, the faster port on the FIFO handles burst transfers while the slower port handles constant rate transfers. However, even with different access types and speeds, the average data rates into and out of the FIFO have to be the same; otherwise, the FIFO overflows or underflows. Like the single-register design, the FIFO holds data in registers or memory while it synchronizes status signals that determine when data can be written into the FIFO or when data can be read out of it.

In speed-matching applications, each port (read or write) has a different clock. The registers in the FIFO use the write-port clock, just as the holding register uses the clock of the circuit changing the register's contents. Signal synchronization takes place in the pointer logic and is more complex than handshake signaling.

Several approaches exist for designing the pointer logic. The first method is to synchronize the read and write strobes while using counters in each clock domain to track the available entries in the FIFO. The counters reflect the number of FIFO entries available for reading or writing, and the counters are synchronous with their respective ports. The read counter tracks the number of entries that contain valid data, and the write counter tracks the number of entries available to store data. When you reset the pointer logic, the read counter starts at zero because no data is available to read. The write counter starts at the number of entries in the FIFO, meaning that all the entries are available for storing data.

The read strobe decrements the read counter and is synchronized to the write-clock domain before it increments the write counter. The write strobe decrements the write counter and is synchro-

nized to the read-clock domain before it increments the read counter.

This design requires single clockwide pulses and pulse synchronizers for the read and write strobes because, as a level signal crosses from one clock domain to a faster one, it remains valid for more clock cycles in the faster domain than in the slow one. Because each counter changes whenever the read or write signal is valid, then the faster clock domain detects more reads or writes than actually occurred in the slower clock domain. Pulse synchronizers translate a clockwide pulse in one clock domain to a clockwide pulse in the new clock domain, and each pulse represents one read or write of the FIFO.

This FIFO-status technique gives pessimistic status for both reads and writes. The status for the write port indicates full when the FIFO has all entries filled and continues to indicate full after the read strobe triggers, because synchronization delays the strobe to the write counter. This situation is also true for the empty status on the read port, because synchronization delays the write strobe to the read counter.

Another consideration for this design is detecting full or empty at the right time. If the FIFO has one entry remaining, and the write strobe triggers, the FIFO must then set the full status. This scenario gives the full indication one clock sooner to allow the circuit writing into the FIFO enough time to stop the next write from overflowing the FIFO. This situation is also true for the read port of the FIFO. In this case, if the FIFO has only one entry, and the read strobe

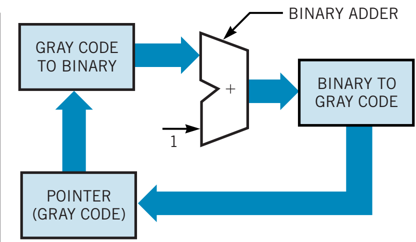


Figure 8 The Gray-code counter, a binary adder, has converters from and to Gray code before and after the adder.

triggers, you must set the empty status to give the read circuitry time to prevent a read of an empty FIFO.

This pointer logic restricts circuits from accessing the FIFO on every clock cycle, even in the slow clock domain. The advantage of this feature is that the circuits accessing the FIFO have at least one clock cycle to evaluate the FIFO status. The FIFO can have every entry filled with data without overwriting valid data or can be empty without reading invalid data. Another advantage of this design is that each side can read its respective counter and determine how many entries are available. A designer can use this FIFO design for circuits that perform multiple reads or writes of data without causing an underflow or overflow condition.

The drawback of this design is that counters, rather than direct comparisons of the read and write pointers, determine status. For large FIFOs, these counters are large. Also, the average data rates are half the slowest clock frequency, because the read or write pulses from the faster clock domain must have at least two clock periods between them in the slow-clock domain when using pulse synchronizers. One way to eliminate some of these problems is to use direct pointer comparison.

In this FIFO design, comparing the read and write pointers determines the FIFO status. Pointer comparison in asynchronous designs is more challenging, because each pointer exists in a different clock domain, and synchronizing a signal bus requires that the bus does not change while synchronizing handshake signals. A FIFO design using this technique for pointer synchronization would be slow. To solve this problem, the FIFO pointer logic uses Gray code instead of binary code for the pointers.

Gray code changes only one bit at a time for each increase or decrease in the count (**Table 3**). You can use synchroniz-

TABLE 3–GRAY CODE VERSUS BINARY CODE

Decimal	Binary	Gray
0	0	0
1	1	1
2	10	11
3	11	10
4	100	110
5	101	111
6	110	101
7	111	100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

ers on Gray-code buses because only one signal changes each time the bus changes, eliminating the race condition between bits of the Gray-coded bus as it passes through separate synchronizers. The pointers for this design are Gray-code counters. Using binary pointers would require synchronizing pointer values after converting them to Gray code, and using conversion logic violates the restriction that synchronized signals originate from flip-flops before crossing a clock domain.

The Gray-code counter is a binary adder with converters from and to Gray code before and after the adder (**Figure 8**). Converting to and from Gray code is an XOR operation, so you need only a few more levels of logic than a binary counter. To convert Gray code to binary, use: $B_N=G_N$; $B_N-1=B_N+G_N-1$;$B_N-2=B_N-1+G_N-2...B_1=B_2+G_1$; $B_0=B_1+G_0$. To convert binary to Gray code, use: $G_N=B_N$; $G_N-1=B_N+B_N-1$;$G_N-2=B_N-1+B_N-2...G_1=B_2+B_1$; $G_0=B_1+B_0$. A design can use the same technique to compare Gray-code pointer values by adding converters between the pointers and binary-comparison logic.

A FIFO with this pointer logic is fast, and circuits can read or write the FIFO on every clock cycle. However, accessing the FIFO on every cycle means the FIFO status has to include almost-full and almost-empty indications, so that the circuits accessing the FIFO have time to stop. Almost full indicates that one entry is available to write, and almost empty indicates that one entry remains unread. This situation describes a design that needs the fewest possible status signals, and a design needs more indicators if the circuits accessing the FIFO use a burst access with a fixed minimum size.

This FIFO-status technique gives pessimistic status for both reads and writes. The status on the write port indicates full when the FIFO fills and continues to indicate full after it the circuit reads it, because synchronization delays the read pointer to the write-side comparison logic. This situation is also true for the empty status on the read side, because synchronization delays the write pointer to the read-side comparison logic.

Handling signals crossing clock domains becomes less challenging when you design with techniques that reduce the risk of communication failure between circuits in different domains. To prevent metastability of flip-flops receiving signals that cross clock domains from causing unpredictable behavior in circuits, use synchronization. Level synchronizers work well for signals that remain valid for many clock cycles. Use an edge-detecting synchronizer for level signals in the slower clock domain that change to pulses in the new clock domain. Finally, use pulse synchronizers for pulses crossing clock domains. Remember that, when a signal bus crosses clock domains, it needs to arrive in the new clock domain during the same clock cycle. Do not synchronize each signal but use a holding register and handshaking. Handshaking indicates when signals in the holding register are valid and when to sample them. Using handshaking and a holding register is useful for data buses but does not provide for passing more than one data word at a time to the new clock domain.□

References

1. Davis, Leroy, *Logic Metastability*, www.interfacebus.com/Design_MetaStable.html, September 2002.
2. Seed, Luke, *Introduction to VLSI/Clocked CMOS Circuits*, The University of Sheffield, UK, www.shef.ac.uk/eee/teach/resources/eee310/documents/VLSI_Clocked_CMOS.pdf, February 2002.
3. Haseloff, Eilhard, "Metastable Response in 5-V Logic Circuits," Texas Instruments Application Note SDYA006, www-s.ti.com/sc/psheets/sdya006/sdya006.pdf, February 1997.
4. Cummings, Clifford E, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," Synopsys Users Group Conference, San Jose, CA, www.sunburst-design.com/papers, March 2001.
5. "A Metastability Primer," Philips Semiconductors Application Note AN2-19, www.semiconductors.philips.com/acrobat/applicationnotes/AN219_1.pdf, Nov 15, 1989.
6. Wakerly, John F, *Digital Design: Principles and Practices*, Prentice Hall, 1990.

Author's biography

Mike Stein is principal consulting engineer at Paradigm Works (Andover, MA), where he works on ASIC and FPGA design, synthesis, and static-timing analysis. He received a BSEE from Northeastern University in 1995. His spare-time interests include playing video games with his son, photography, snowboarding, biking, and building Lego robots.