

8.9.8 Synchronizing High-Speed Data Transfers

A very common problem in computer systems is synchronizing external data transfers with the computer system clock. A simple example is the interface between personal computer's network interface card and a 100 Mbps Ethernet link. The interface card may be connected to a PCI bus, which has a 33.33 MHz clock. Even though the Ethernet speed is an approximate multiple of the bus speed, the signal received on the Ethernet link is generated by another computer whose transmit clock is not synchronized with the receive clock in any way. Yet the interface must still deliver data reliably to the PCI bus.

Figure 8-102 shows the problem. NRZ serial data RDATA is received from the Ethernet at 100 Mbps. The digital phase-locked loop (DPLL) recovers a 100-MHz clock signal RCLK which is centered on the 100 Mbps data stream and

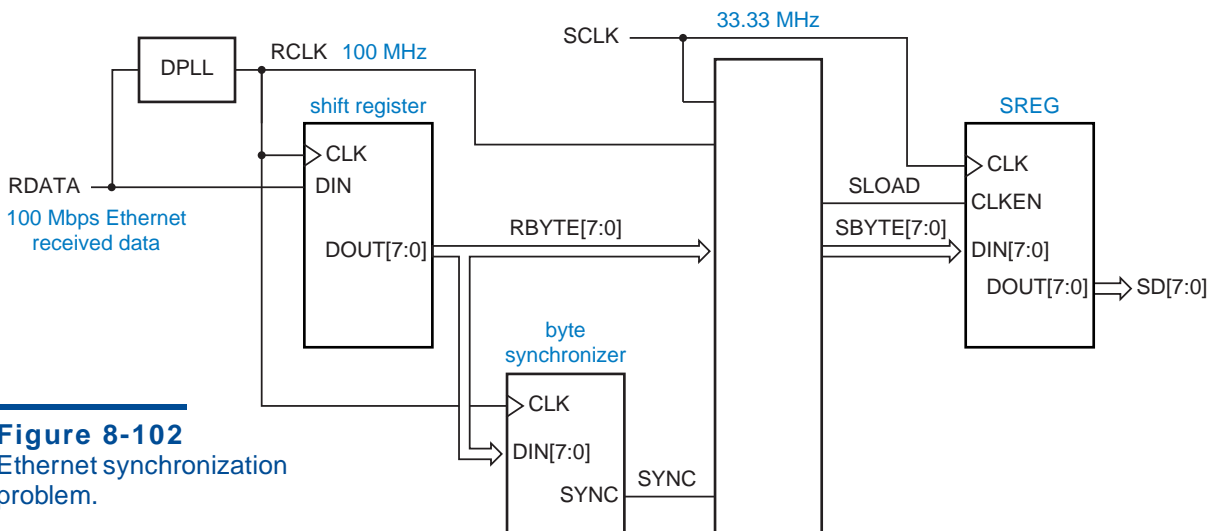


Figure 8-102
Ethernet synchronization problem.

ONE NIBBLE AT A TIME

The explanation of 100 Mbps Ethernet reception above is oversimplified, but sufficient for discussing the synchronization problem. In reality, the received data rate is 125 Mbps, where each 4 bits of user data is encoded as a 5-bit symbol using a so-called 4B5B code. By using only 16 out of 32 possible 5-bit codewords, the 4B5B code guarantees that regardless of the user data pattern, the bit stream on the wire will have a sufficient number of transitions to allow clock recovery. Also, the 4B5B code includes a special code that is transmitted periodically to allow nibble (4-bit) and byte synchronization to be accomplished very easily.

As a result of nibble synchronization, a typical 100-Mbps Ethernet interface does not see an unsynchronized 100 MHz stream of bits. Instead, it sees an unsynchronized 25 MHz stream of nibbles. So, the details of a real 100-Mbps Ethernet synchronizer are different, but the same principles apply.

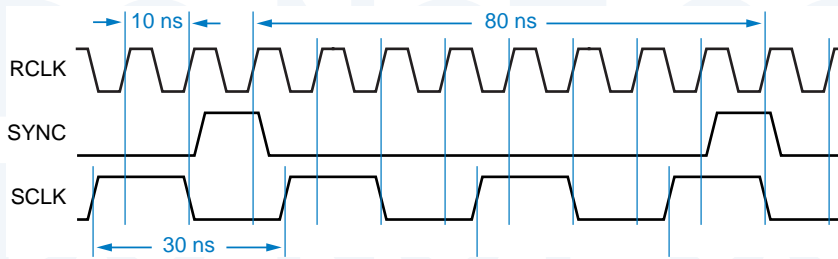


Figure 8-103
Ethernet link and system clock timing.

allows data to be clocked bit-by-bit into an 8-bit shift register. At the same time, a byte synchronization circuit searches for special patterns in the received data stream that indicate byte boundaries. When it detects one of these, it asserts the SYNC signal and does so on every eighth subsequent RCLK tick, so that SYNC is asserted whenever the shift register contains an aligned 8-bit byte. The rest of the system is clocked by a 33.33 MHz clock SCLK. We need to transfer each aligned byte RBYTE[7:0] into a register SREG in SCLK’s domain. How can we do it?

Figure 8-103 shows some of the timing. We immediately see is that the byte-aligned signal, SYNC, is asserted for only 10 ns per byte. We have no hope of consistently detecting this signal with the asynchronous SCLK, whose period is a much longer 30 ns.

The strategy that is almost universally followed in this kind of situation is to transfer the aligned data first into a holding register HREG in the *receive* clock (RCLK) domain. This gives us a lot more time to sort things out, 80 ns in this case. Thus, the “?” box in Figure 8-102 can be replaced by Figure 8-104, which shows HREG and a box marked “SCTRL.” The job of SCTRL is to assert SLOAD during exactly one 30-ns SCLK period, so that the output of HREG is valid and stable for the setup and hold times of register SREG in the SCLK domain. SLOAD also serves as a “new-data available” signal for the rest of the interface, indicating that a new data byte will appear on SBYTE[7:0] during the

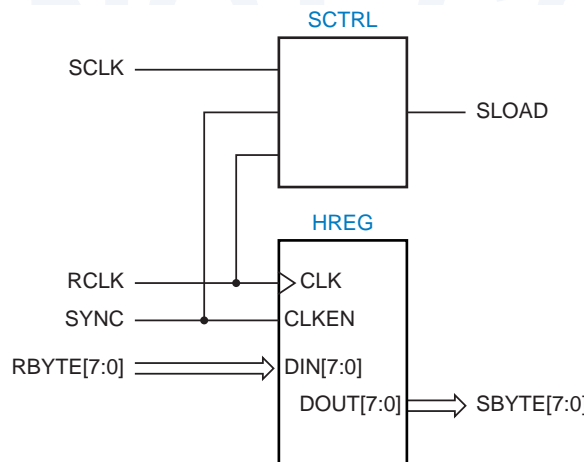


Figure 8-104
Byte holding register and control.

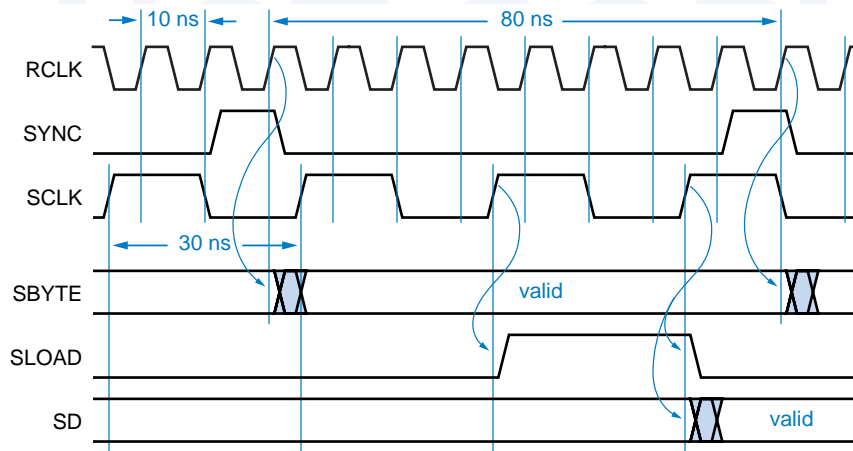


Figure 8-105
Timing for SBYTE
and possible timing
for SLOAD.

next SCLK period. Figure 8-105 shows possible timing for SLOAD based on this approach and the previous timing diagram.

Figure 8-106 is a circuit that can generate SLOAD with the desired timing. The idea is to use SYNC to set an S-R latch as a new byte becomes available. The output of this latch, NEWBYTE, is sampled by FF1 in the SCLK domain. Since NEWBYTE is not synchronized with SCLK, FF1’s output SM may be metastable, but it is not used by FF2 until the next clock tick, 30 ns later. Assuming that the AND gate is reasonably fast, this gives plenty of metastability resolution time. FF2’s output is the SLOAD signal. The AND gate ensures that SLOAD is only one SCLK period wide; if SLOAD is already 1, it can’t be set to 1 on the next tick. This gives time for the S-R latch to be reset by SLOAD in preparation for the next byte.

A timing diagram for the overall circuit with “typical” timing is shown in Figure 8-107. Since SCLK is asynchronous to RCLK and SYNC, it can have an arbitrary relationship with RCLK and SYNC. In the figure, we’ve shown a case where the next SCLK rising edge occurs well after NEWBYTE is set. Although the figure shows a window in which SM and SM1 could be metastable in the general case, metastability doesn’t actually happen when the timing is as drawn. Later, we’ll show what can happen if the SCLK edge occurs when NEWBYTE is changing.

We should make several notes about the circuit in Figure 8-106. First, the SYNC signal must be glitch-free, since it controls the S input of a latch, and it must be wide enough to meet the minimum pulse width requirement of the latch.

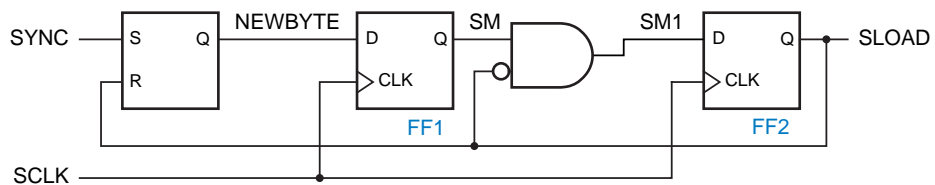


Figure 8-106
SCTRL circuit for
generating SLOAD.

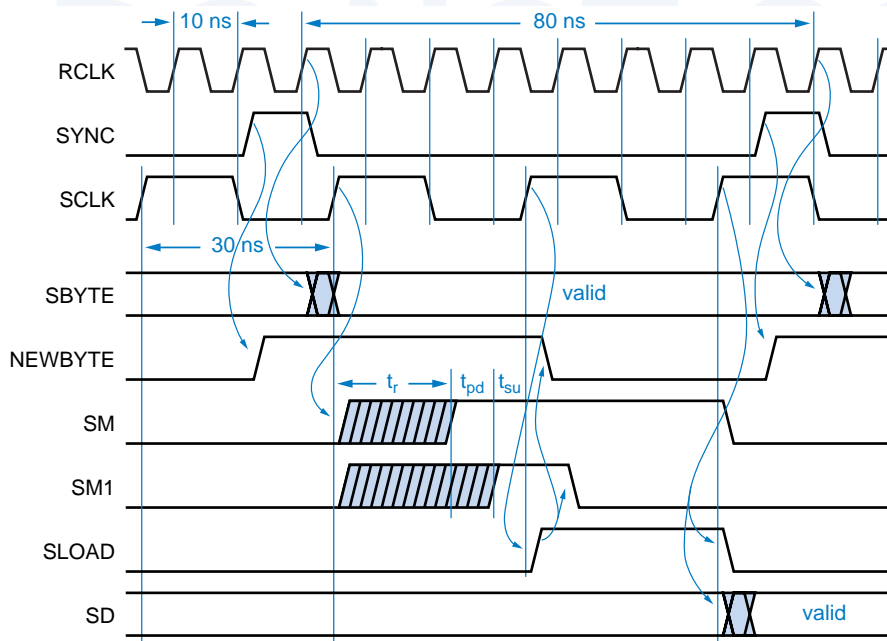


Figure 8-107
Timing for the SCTRL
circuit in Figure 8-106.

Since the latch is set on the leading edge of SYNC, we actually cheated a little; NEWBYTE may be asserted a little *before* a new byte is actually available in HREG. This is OK, because we know that it takes two SCLK periods from when NEWBYTE is sampled until SREG is loaded. In fact, we might have cheated even more if an earlier version of SYNC was available (see Exercise 8.76).

Assuming that t_{su} is the setup time of a D flip-flop and t_{pd} is the propagation delay of the AND gate in Figure 8-106, the available metastability resolution time t_r is one SCLK period, 30 ns, minus $t_{su} + t_{pd}$, as shown in Figure 8-107. The timing diagram also shows why we can't use SM directly as the reset signal for the S-R latch. Since SM can be metastable, it could wreak havoc. For example, it could be semi-HIGH long enough to reset the latch but then fall back to LOW; in that case, SLOAD would not get set and we would miss a byte. By using instead the output of the synchronizer (SLOAD) both for the latch reset and for the load signal in the SCLK domain, we ensure that the new byte is detected and handled consistently in both clock domains.

The timing that we showed in Figure 8-107 is nominal, but we also have to analyze what happens if SCLK has a different phase relationship with RCLK and SYNC than what is shown. You should be able to convince yourself that if the SCLK edge occurs earlier, so that it sample NEWBYTE just as it's going HIGH, everything still works as before, and the data transfer just finishes a little sooner. The more interesting case is when SCLK occurs later, so that it just misses NEWBYTE as it's going HIGH, and catches it one SCLK period later. This timing is shown in Figure 8-108.

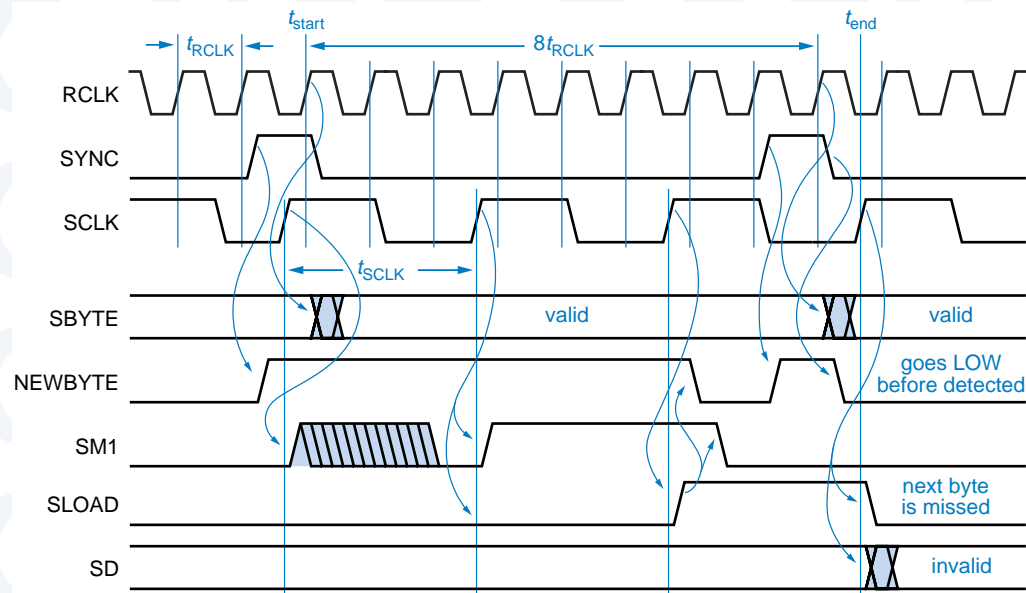


Figure 8-108 Maximum-delay timing for SCTRL circuit.

In the timing diagram, we have shown NEWBYTE going high around the same time as the SCLK edge—less than FF1's t_{su} before the edge. Thus, FF1 may not see NEWBYTE as HIGH or its output may become metastable, and it does not solidly capture NEWBYTE until one SCLK period later. Two SCLK periods after that, we get the SCLK edge that loads SBYTE into SREG.

This timing scenario is bad news, because by the time the load occurs, SBYTE is already changing to the *next* received byte. In addition, SLOAD happens to be asserted during and a little bit after the SYNC pulse for this next received byte. Thus, the latch has both S and R asserted simultaneously. If they are removed simultaneously, the latch output may become metastable. Or, as we've shown in the timing diagram, if NEWBYTE (R) is negated last, then the latch is left in the reset state, and this next received byte is never detected and loaded into the SCLK domain.

Thus, we need to analyze the maximum-delay timing case carefully to determine if a synchronizer will work properly. Figure 8-108 shows a starting reference point t_{start} for the SCTRL circuit, namely the RCLK edge on which a byte is loaded into HREG, at end of SYNC pulse). The ending reference point t_{end} is the SCLK edge on which SBYTE is loaded into SREG. The maximum delay between these two reference points, which we'll call t_{maxd} , is the sum of the following components:

$-t_{RCLK}$ Minus one RCLK period, the delay from t_{start} back to the edge on which SYNC was asserted. This number is negative because SYNC is asserted one clock tick before the tick that actually loads HREG.

- t_{CQ} One flip-flop CLK-to-Q maximum delay. Assuming that SYNC is a direct flip-flop output in the RCLK domain, this is delay from the RCLK edge until SYNC is asserted.
- t_{SQ} Maximum delay from S to Q in the S-R latch in Figure 8-106. This is the delay for NEWBYTE to be asserted.
- t_{su} Setup time of FF1 in Figure 8-106. NEWBYTE must be asserted at or before the setup time to guarantee detection.
- t_{SCLK} One SCLK period. Since RCLK and SCLK are asynchronous, there may be a delay of up to one SCLK period before the next SCLK edge comes along to sample NEWBYTE.
- t_{SCLK} After NEWBYTE is detected by FF1, SLOAD is asserted on the next SCLK tick.
- t_{SCLK} After SLOAD is asserted, SBYTE is loaded into SREG on the next SCLK tick.

Thus, $t_{maxd} = 3t_{SCLK} + t_{CQ} + t_{SQ} + t_{su} - t_{RCLK}$. A few other parameters must be defined to complete the analysis:

- t_h The hold time of SREG.
- $t_{CQ(min)}$ The minimum CLK-to-Q delay of HREG, conservatively assumed to be 0.
- t_{rec} The recovery time of the S-R latch, the minimum time allowed between negating S and negating R (see box on page 441).

To be loaded successfully into SREG, SBYTE must remain valid until at least time $t_{end} + t_h$. The point at which SBYTE changes and becomes invalid is 8 RCLK periods after t_{start} , plus $t_{CQ(min)}$. Thus, for proper circuit operation we must have

$$t_{end} + t_h \leq t_{start} + 8t_{RCLK}$$

For the maximum-delay case, we substitute $t_{end} = t_{start} + t_{maxd}$ into this relation and subtract t_{start} from both sides to obtain

$$t_{maxd} + t_h \leq 8t_{RCLK}$$

Substituting the value of t_{maxd} and rearranging, we obtain

$$3t_{SCLK} + t_{CQ} + t_{SQ} + t_{su} + t_h \leq 9t_{RCLK} \quad (8-1)$$

as the requirement for correct circuit operation. Too bad. Even if we assume very short component delays (t_{CQ} , t_{SQ} , t_{su} , t_h), we know that $3t_{SCLK}$ (90 ns) plus anything is going to be more than $9t_{RCLK}$ (also 90 ns). So this design will never work properly in the maximum-delay case.

Even if the load-delay analysis gave a good result, we would still have to consider the requirements for proper operation of the SCTRL circuit itself. In particular, we must ensure that when the SYNC pulse for the next byte occurs, it

is not negated until time t_{rec} after SLOAD for the previous byte was negated. So, another condition for proper operation is

$$t_{\text{end}} + t_{\text{CQ}} + t_{\text{rec}} \leq t_{\text{start}} + 8t_{\text{RCLK}} + t_{\text{CQ}(\text{min})}$$

Substituting and simplifying as before, we get another requirement that isn't met by our design:

$$3t_{\text{SCLK}} + 2t_{\text{CQ}} + t_{\text{SQ}} + t_{\text{su}} + t_{\text{rec}} \leq 9t_{\text{RCLK}} \quad (8-2)$$

There are several ways that we can modify our design to satisfy the worst-case timing requirements. Early in our discussion, we noted that we “cheated” by asserting SYNC one RCLK period before the data in HREG is valid, and that we actually might get away with asserting SYNC even soon. Doing this can help us meet the maximum delay requirement, because it reduces the “ $8t_{\text{RCLK}}$ ” term on the right-hand side of the relations. For example, if we asserted SYNC two RCLK periods earlier, we would reduce this term to “ $6t_{\text{RCLK}}$ ”. However, there's no free lunch, we can't assert SYNC arbitrarily early. We must also consider a *minimum delay* case, to ensure that the new byte is actually available in HREG when SBYTE is loaded into SREG. The minimum delay t_{mind} between t_{start} and t_{end} is the sum of the following components:

$-nt_{\text{RCLK}}$ Minus n RCLK periods, the delay from t_{start} back to the edge on which SYNC was asserted. In the original design, $n = 1$.

$t_{\text{CQ}(\text{min})}$ This is the minimum delay from the RCLK edge until SYNC is asserted, conservatively assumed to be 0.

t_{SQ} This is the delay for NEWBYTE to be asserted, again assumed to be 0.

$-t_{\text{h}}$ Minus the hold time of FF1 in Figure 8-106. NEWBYTE might be asserted at the end of the hold time and still be detected.

$0t_{\text{SCLK}}$ Zero times the SCLK period. We might get “lucky” and have the SCLK edge come along just as the hold time of FF1 is ending.

t_{SCLK} A one-SCLK-period delay to asserting SLOAD, as before.

t_{SCLK} A one-SCLK-period delay to loading SBYTE into SREG, as before.

In other words, $t_{\text{mind}} = 2t_{\text{SCLK}} - t_{\text{h}} - nt_{\text{RCLK}}$.

For this case, we must ensure that the new byte has propagated to the output of HREG when the setup time window of SREG begins, so we must have

$$t_{\text{end}} - t_{\text{su}} \geq t_{\text{start}} + t_{\text{co}},$$

where t_{co} is the maximum clock-to-output delay of HREG. Substituting $t_{\text{end}} = t_{\text{start}} + t_{\text{mind}}$ and subtracting t_{start} from both sides, we get

$$t_{\text{mind}} - t_{\text{su}} \geq t_{\text{co}}.$$

Substituting the value of t_{mind} and rearranging, we get the final requirement,

$$2t_{\text{SCLK}} - t_{\text{h}} - t_{\text{su}} - t_{\text{co}} \geq nt_{\text{RCLK}} \quad (8-3)$$

If, for example, t_h , t_{su} , and t_{co} are 10 ns each, the maximum value of n is 3; we can't generate SYNC more than two clock ticks before its original position in Figure 8-108. This may or may not be enough to solve the maximum-delay problem, depending on other delay values; this is explored for a particular set of components in Exercise 8.76.

Moving the SYNC pulse earlier may not give enough delay improvement or may not be an available option in some systems. An alternative solution that can always be made to work is to increasing the time between successive data transfers from one clock domain to the other. We can always do this because we can always transfer more bits per synchronization. In the Ethernet-interface example, we could collect 16 bits at a time in the RCLK domain and transfer 16 bits at a time to the SCLK domain. This changes the previously stated $8t_{RCLK}$ terms to $16t_{RCLK}$, providing a lot more margin for the maximum-delay timing requirements. Once 16 bits have been transferred into the SCLK domain, we can still break them into two 8-bit chunks if we need to process the data a byte at a time.

It may also be possible to improve performance by modifying the design of the SCTRL circuit. Figure 8-111 shows a version where SLOAD is generated directly by the flip-flop that samples NEWBYTE. In this way, SLOAD appears one SCLK period sooner than in our original SCTRL circuit. Also, the S-R latch is cleared sooner. This circuit works only if a couple of key assumptions are true:

1. A reduced metastability resolution time for FF1 is acceptable, equal to the time that SCLK is HIGH. Metastability must be resolved before SCLK goes LOW, because that's when the S-R latch gets cleared if SLOAD is HIGH.
2. The setup time of SREG's CLKEN input (Figure 8-102) is less than or equal to the time that SCLK is LOW. Under the previous assumption, the SLOAD signal applied to CLKEN might be metastable until SCLK goes LOW.
3. The time that SCLK is LOW is long enough to generate a reset pulse on RNEW that meets the minimum pulse-width requirement of the S-R latch.

Note that these behaviors makes proper circuit operation dependent on the duty cycle of SCLK. If SCLK is relatively slow and its duty cycle is close to 50%, this circuit generally works fine. But if SCLK is too fast or has a very small, very large, or unpredictable duty cycle, the original circuit approach must be used.

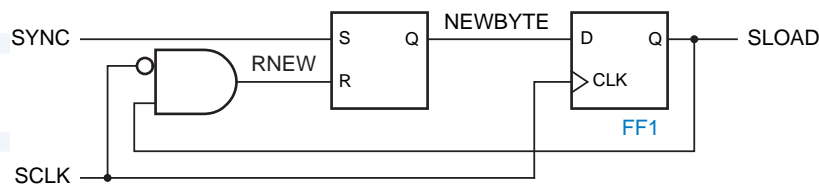


Figure 8-109
Half-clock-period
SCTRL circuit for
generating SLOAD.

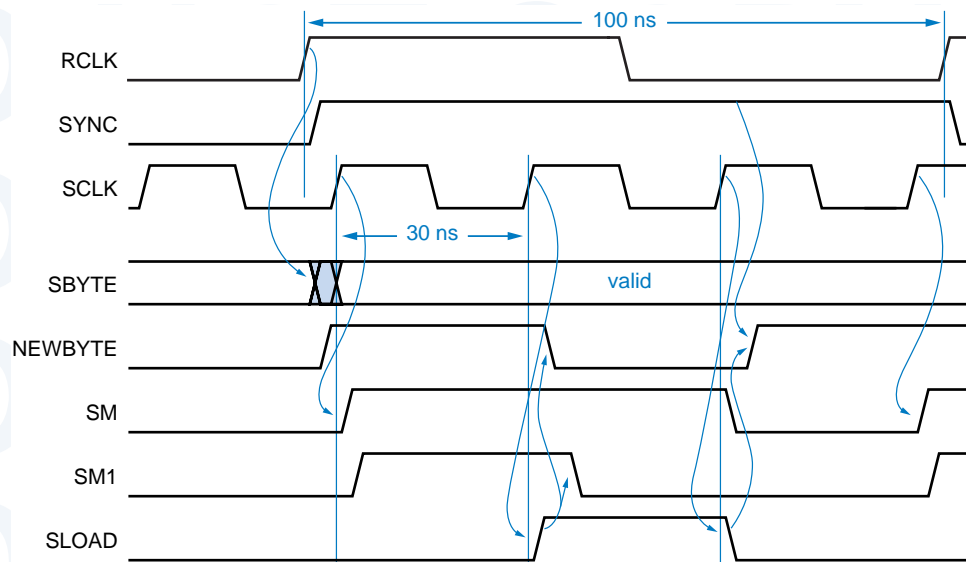
All of these synchronization schemes require the clock frequencies to be within a certain range of each other for proper circuit operation. This must be considered for testing, where the clocks are usually run slower, and for upgrades, where one or both clocks may run faster. For example, in the Ethernet interface example, we wouldn't change the frequency of standard 100-Mbps Ethernet, but we might upgrade the PCI bus from 33 to 66 MHz.

The problems caused by clock frequency changes can be subtle. To get a better handle on what can go wrong, it's useful to consider how a synchronizer works (or doesn't work!) if one clock frequency changes by a factor of 10 or more.

For example what happens to the synchronizer timing in Figure 8-107 if we change RCLK from 100 MHz to 10 MHz? At first glance, it would seem that all is well, since a byte now arrives only once every 800 ns, giving much more time for the byte to be transferred into the SCLK domain. Certainly, Eqn. 8-1 on page 681 and Eqn. 8-2 on page 682 are satisfied with much more margin. However, Eqn. 8-3 is no longer satisfied unless we reduce n to zero! This could be accomplished by generating SYNC one RCLK tick later than is shown in Figure 8-107.

But even with this change, there's *still* a problem. Figure 8-110 shows the new timing, including the later SYNC pulse. The problem is that the SYNC pulse is now 100 ns long. As before, NEWBYTE (the output of the S-R latch in

Figure 8-110 Synchronizer timing with slow (10 MHz) RCLK.



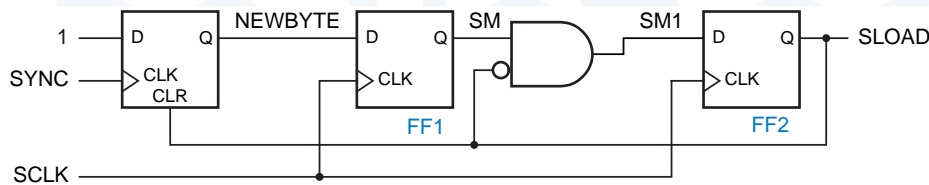


Figure 8-111
Synchronizer with
edge-triggered
SYNC detection.

Figure 8-106 on page 678) is set by SYNC and is cleared by SLOAD. The problem is that when SLOAD goes away, SYNC is still asserted, as shown in the new timing diagram. Thus, the new byte will be detected and transferred twice!

The solution to the problem is to detect only the leading edge of SYNC, so that the circuit is not sensitive to the length of the SYNC pulse. A common way of doing this is to replace the S-R latch with an edge-triggered D flip-flop, as shown in Figure 8-111. The leading edge of SYNC sets the flip-flop, while SLOAD is used as an asynchronous clear as before.

The circuit in Figure 8-111 solves the slow-RCLK problem, but it also changes the derivation of Eqns. 8-1 through 8-3 and may make timing more constrained in some areas (see Exercise 8.77). Another disadvantage that this circuit cannot be realized in a typical PLD, which has all flip-flops controlled by the same clock; instead, a discrete flip-flop must be used to detect SYNC.

After reading almost 10 pages to analyze just one “simple” example, you should have a strong appreciation of the difficulty of correct synchronization-circuit design. Several guidelines can help you:

- Minimize the number of different clock domains in a system.
- Clearly identify all clock boundaries and provide clearly identified synchronizers at those boundaries.
- Provide sufficient metastability resolution time for each synchronizer so that synchronizer failure is rare, much more unlikely than other hardware failures.
- Analyze synchronizer behavior over a range of timing scenarios, including faster and slower clocks that might be applied as a result of system testing or upgrades.
- Simulate system behavior over a wide range of timing scenarios as well.

The last guideline above is a catch-all for modern digital designers, who usually rely on sophisticated, high-speed logic simulators to find their bugs. But it’s not a substitute for following the first four guidelines. Ignoring them can lead to problems that cannot be detected by a typical, small number of simulation scenarios. Of all digital circuits, synchronizers are the ones for which it’s most important to be “correct by design”!