A-PDF Split DEMO : Purchase from www.A-PDF.com to remove the watermark

CHAPTER 16

CLOCK AND SYNCHRONIZATION: PRINCIPLE AND PRACTICE

The single most important design principle used in this book is the synchronous methodology, in which all registers are controlled by a common clock signal. Design and analysis so far are based on an ideal clocking scenario. We assume that the entire system can be driven by a single clock signal and that the sampling edge of this clock signal can reach all registers at the same time. In reality, this is hardly possible. We need to take into consideration a non-ideal clock signal and sometimes even have to divide a large system into subsystems with independent clock signals. This chapter discusses the modeling and effect of a non-ideal clock signal, the synchronization of an asynchronous signal, and the interface between two independent clock domains.

16.1 OVERVIEW OF A CLOCK DISTRIBUTION NETWORK

16.1.1 Physical implementation of a clock distribution network

The clock distribution network is the circuit that distributes the clock signal to all FFs in the system. Since the circuit does not perform any logic function, its design and analysis are mainly at the transistor level. In Section 6.5.1, we discussed the low-level model of gates and wires for propagation delay calculation. As shown in Figure 6.15, each input port of a gate and each wire introduce small values of resistance and capacitance. The output port of a cell has to charge or discharge (i.e., "drive") all capacitors when a signal switches state. The number of input ports driven by a cell is known as *fan-out*. The driving capability of

Copyright © 2006 John Wiley & Sons, Inc.

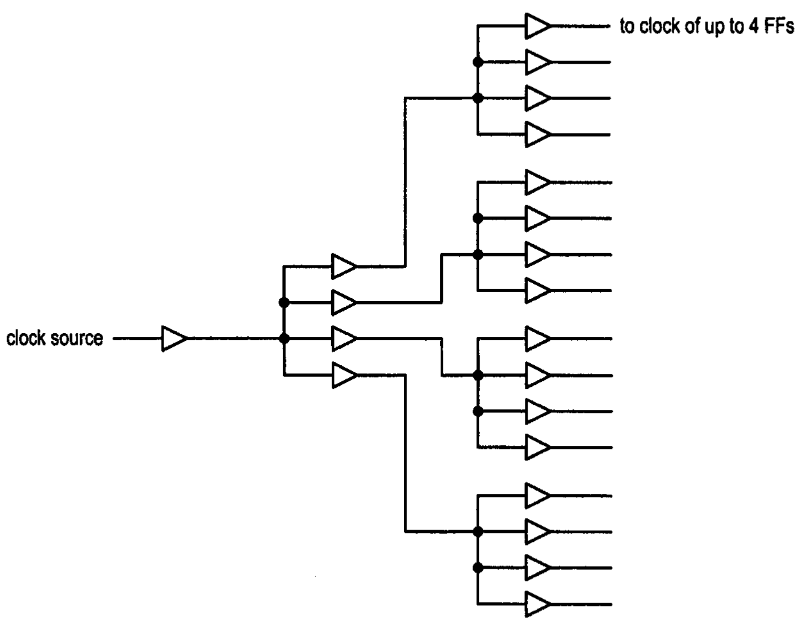


Figure 16.1 Conceptual clock distribution network.

a cell depends on the electrical characteristics of the internal transistors. A typical cell can normally drive up to half a dozen cells.

While the basic transistor-level model of the clock distribution network is similar to that in Figure 6.15, the fan-out is much larger. Since all registers are connected to the same clock signal in a synchronous circuit, the fan-out of a clock signal is the number of FFs in the system. It may reach thousands or even tens of thousands in a large system. Thus, the physical implementation of a clock distribution network is very different from that of regular connection wires. Its construction is separated from the routing of regular logic and processed independently.

In addition to the clock signal, the reset signal is connected to all FFs of the system. Thus, construction of the reset network is somewhat similar to that of the clock distribution network. Because the reset signal does not impose many strict timing constraints, its implementation is simpler and less critical.

Clock synthesis of ASIC devices In ASIC technology, the clock distribution network is constructed by a process known as *clock synthesis*, which is a step in the physical design. The clock synthesis uses multiple levels of buffers to increase the driving capability and applies a special routing algorithm to balance the distribution network and minimize the difference in propagation delays. A conceptual three-level clock distribution network is shown in Figure 16.1. We assume that each buffer can drive four input ports. The buffers are used to increase the driving capability and do not perform any logic function.

An example of idealized physical routing of the previous distribution network is shown in Figure 16.2. It is done by a two-level recursive H-shaped network so that the wire length from the clock source to each FF is about the same. While the propagation delay from the clock source to an FF is unavoidable, this routing helps to ensure that the clock signal reaches each FF at about the same time.

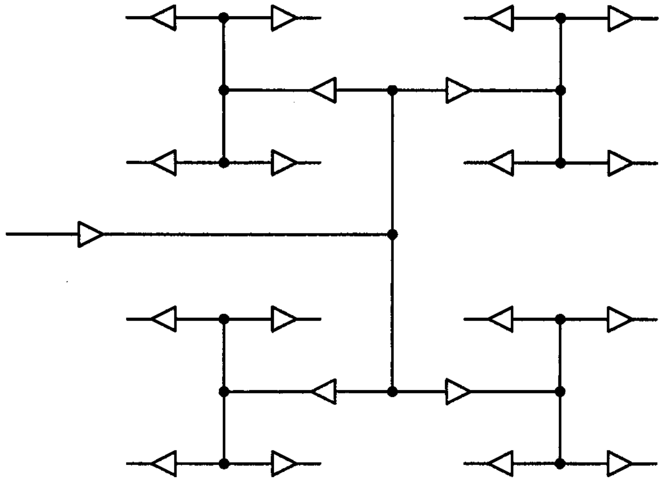


Figure 16.2 Idealized routing of a clock distribution network.

Clock distribution networks of FPGA devices In FPGA technology, a chip usually has one or more prerouted and prefabricated clock distribution networks. If we develop the VHDL code in a disciplined way, the synthesis software can recognize the existence of the clock signal and automatically map it to a prefabricated clock distribution network.

16.1.2 Clock skew and its impact on synchronous design

The construction and analysis of a clock distribution network is essentially a task at the transistor level. At the gate and RT levels, the effect of the clock distribution network is modeled by propagation delays from the clock source to various registers. Because of the variation in buffering and routing, the propagation delays may be different, as shown in the simple example in Figure 16.3. The key characteristic is the difference between the arrival times of the sampling edges, which is known as the *clock skew*. For multiple registers, we consider the worst-case scenario and define the clock skew as the difference between the arrival times of the earliest and latest sampling edges.

As the size of a circuit and the number of FFs increase, the clock distribution network becomes larger and more complex. Controlling the arrival time of the clock's sampling edge to each FF becomes more difficult. This introduces larger variations over the arrival times, which, in turn, increase the clock skew. Thus, we can expect that the clock skew increases with the size of the circuit.

To accommodate the existence of clock skew, we have to modify the synchronous design methodology. The modification depends on the size and clock rate of the system. For a small circuit, propagation delays from the clock source to various FFs are small and almost identical, which implies that the rising edge of the clock signals arrives at the register at almost the same time. We can treat this as the ideal clocking scenario and ignore the clock skew.

For a moderately-sized system, the clock skew is normally a small fraction (a few percent) of the clock period. We can treat it as an ideal synchronous system and design it accordingly. However, during the analysis of setup time and hold time constraints, the

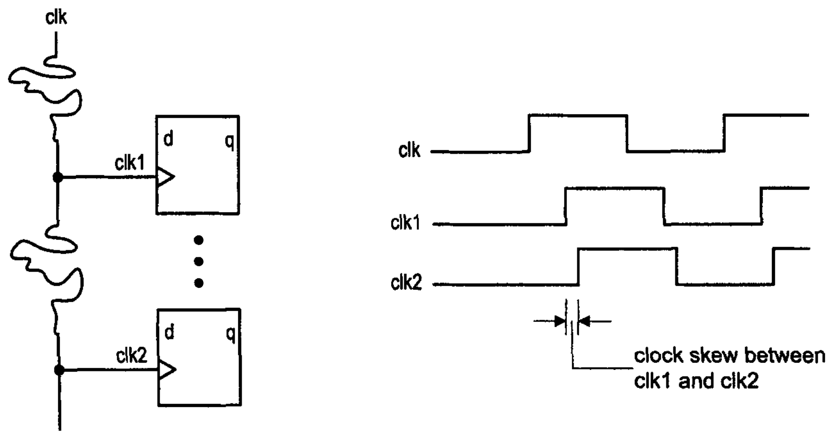


Figure 16.3 Clock skew.

clock skew must be taken into consideration. The skew usually introduces tighter timing requirements and reduces system performance. Current technology can support a clock distribution network with up to several tens of thousands of FFs with an acceptable clock skew. Recall that the proper partition size for synthesis is between 5000 and 50,000 gates. There are no problems applying synchronous design methodology inside each partition block. Section 16.2 discusses the effect of small clock skew in timing analysis.

For a fast, large-scale system, the skew may become comparable to the clock period and can no longer be treated as a small deviation of the arrival time. Because of the lack of a common clock, the synchronous design methodology can no longer be applied. One way to deal with this problem is to divide the system into several smaller subsystems and let each subsystem be controlled by an independent clock signal. Whereas the internal operation of a subsystem is synchronous, its interface to other subsystems is asynchronous. Because of the asynchronous interface, timing violations may occur. We need to use special synchronization schemes and protocols to ensure that the control signals and data can be transferred between subsystems reliably. These schemes and protocols are discussed in Sections 16.3 to 16.9.

16.2 TIMING ANALYSIS WITH CLOCK SKEW

Clock skew is the difference between the arrival times of the sampling edges of a clock signal. It has a significant impact on the timing of sequential circuits. In general, clock skew degrades system performance and imposes a tighter hold time margin. In Section 8.6, we analyzed the timing of sequential circuits with an ideal clock and showed conditions to meet setup time and hold time constraints. The following subsections repeat the analysis with the existence of clock skew.

16.2.1 Effect on setup time and maximal clock rate

For a digital system with an ideal clock, there is no clock skew. We bundle all registers together into a single element, as shown in the basic sequential circuit block diagram of Figure 8.5. To express the different arrival times, individual registers must be considered.

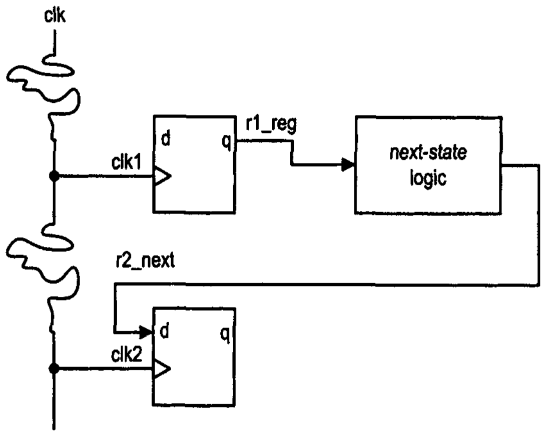


Figure 16.4 Next-state logic feedback path with positive clock skew.

A conceptual diagram of two registers and a single feedback path is shown in Figure 16.4. We assume that the lengthy routing wire introduces a delay of T_{skew} and thus the arrival times of the rising edges are different for the clk1 and clk2 signals. In this particular case, the arrival time of the clk2 signal is late by the amount T_{skew}. The late arrival is also known as a *positive* clock skew.

The timing diagram is shown in Figure 16.5. We follow the procedure used in Section 8.6.2 to analyze the new circuit. To satisfy the setup time constraint, the r2_next signal must be stabilized before t_4. This requirement translates into the condition

$$t_3 < t_4$$

From the timing diagram, we see that

$$t_3 = t_0 + T_{cq} + T_{next(max)}$$

and

$$t_4 = t_5 - T_{setup} = (t_0 + T_c + T_{skew}) - T_{setup}$$

After we substitute t_3 and t_4, the inequality equation is simplified to

$$T_{cq} + T_{next(max)} + T_{setup} - T_{skew} < T_c$$

and the minimal clock period is

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup} - T_{skew}$$

Note that in this particular case, the existence of clock skew reduces the minimal clock period and actually helps the performance.

The clock skew does not always mean late arrival of the sampling edge. For example, if we switch the locations of the two registers in the previous example, the arrival time of the clk2 signal is ahead of the arrival time of the clk1 signal by the amount T_{skew}. The early arrival is also known as a *negative* clock skew. In this case, t_4 becomes

$$t_4 = t_5 - T_{setup} = (t_0 + T_c - T_{skew}) - T_{setup}$$

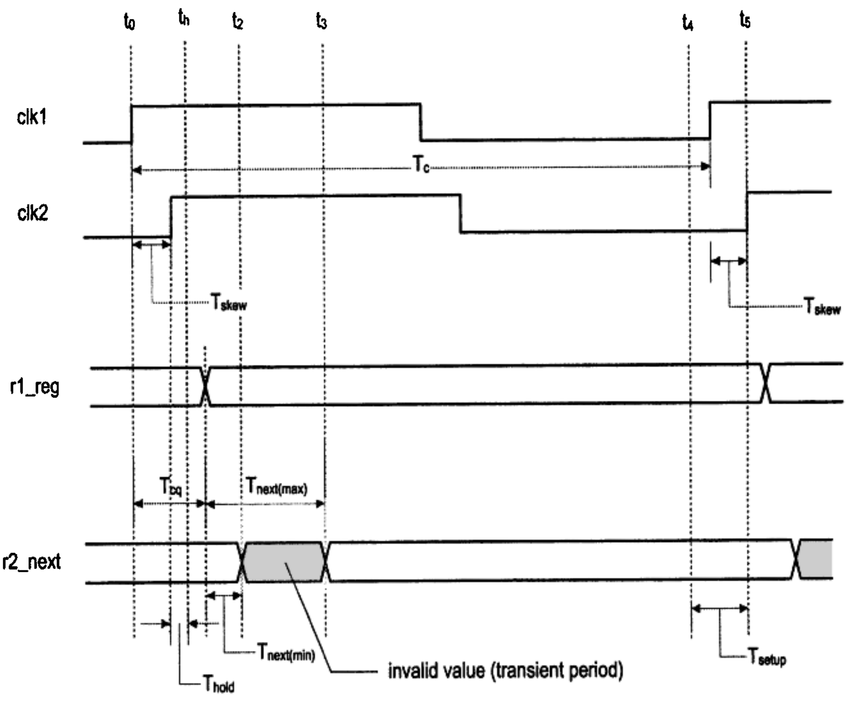


Figure 16.5 Timing analysis of positive clock skew.

and the minimal clock period becomes

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup} + T_{skew}$$

This implies that the clock period must be increased by the amount T_{skew}.

If there are multiple feedback paths, the effect of the clock skew can be positive for some paths and negative for the others. Consider that we add a feedback path from the r2 register to the r1 register, as shown in Figure 16.6. For simplicity, we assume that the propagation delays of the next-state logic are identical. The minimal clock periods of the two paths are

$$T_{cq} + T_{next(max)} + T_{setup} - T_{skew}$$

and

$$T_{cq} + T_{next(max)} + T_{setup} + T_{skew}$$

respectively. Since the design has to satisfy the worst-case scenario, the clock period of the system must be at least $T_{cq} + T_{next(max)} + T_{setup} + T_{skew}$.

While the positive clock skew can be used to reduce the minimal clock period in theory, it is very difficult to do this in real design because of the existence of multiple feedback paths and constraints on the placement of registers and routing of the clock distribution network. The clock skew usually has a negative effect on the clock period and thus imposes a penalty on performance.

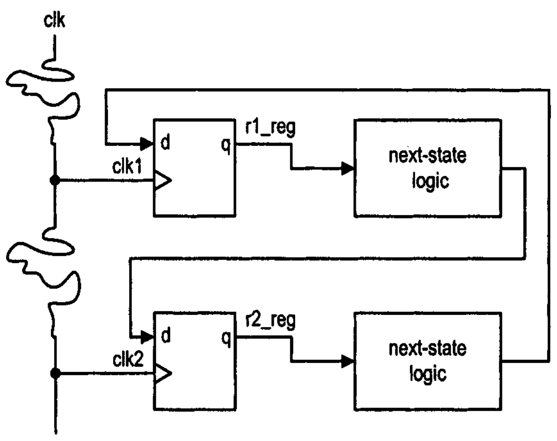


Figure 16.6 Circuit with multiple feedback paths.

16.2.2 Effect on hold time constraint

The hold time analysis is similar to that in Section 8.6.3. To satisfy the hold time constraint, we must ensure that

$$t_h < t_2$$

From the timing diagram, we see that

$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

and

$$t_h = t_0 + T_{hold} + T_{skew}$$

After substitution, the inequality equation can be simplified to

$$T_{hold} < T_{cq} + T_{next(min)} - T_{skew}$$

Compared to the original inequality equation, the positive clock skew imposes a tighter margin for the hold time constraint.

In the worst-case scenario, $T_{next(min)}$ can be close to 0 (when the output of one FF is connected to the input of another FF, as in a shift register). The inequality equation becomes

$$T_{hold} < T_{cq} - T_{skew}$$

Recall that the device manufacture usually guarantees that $T_{hold} < T_{cq}$, and thus the inequality equation will always be satisfied if there is no clock skew. With a positive clock skew, the margin on the inequality equation will be reduced. It may even lead to a timing violation if T_{skew} is greater than the safety margin of $T_{cq} - T_{hold}$.

Unfortunately, there is no fix from the RT-level design. T_{hold}, T_{cq} and T_{skew} are the three parameters in the inequality equation. The first two are inherent characteristics of the device, and the third can only be obtained after synthesis of the clock distribution network. Although in theory we can add some redundant combinational logic (such as a pair of cascading invertors) to introduce artificial propagation delay, this approach is delay sensitive and is not recommended. This problem is better left for the physical design. After construction of the clock distribution network and completion of the placement and routing, accurate clock skew information can be obtained. The physical design software will check for hold time violations. It should correct the problem by rearranging the clock routing or inserting a delay element in the violated path.

The analysis of negative clock skew is similar. The inequality equation becomes

$$T_{hold} < T_{cq} + T_{next(min)} + T_{skew}$$

The clock skew provides extra slack. Since the device manufacturer usually guarantees that $T_{hold} < T_{cq}$, the extra margin provides no additional benefit.

16.3 OVERVIEW OF A MULTIPLE-CLOCK SYSTEM

When we apply the synchronous design methodology, all FFs of the system are controlled by a single global clock. However, as digital systems grow more complex, it becomes very difficult or even impossible to follow the pure synchronous design principle. Multiple clocks may exist or become necessary for several reasons:

- *Inherent multiple-clock sources.* A digital system frequently needs to interact with external systems, such as peripheral devices, or to exchange information through communication links. These external systems or links may not use the same clock signal.
- *Circuit size.* As discussed in Section 16.1, the clock skew increases with the size of the circuit and the number of FFs. When a circuit is large, it is not possible to maintain a single clock. We must divide the system into smaller subsystems and use separate clock signals in the subsystems.
- *Design complexity.* A large digital system is frequently composed of several small subsystems of different performance and power criteria. Applying pure synchronous design methodology may introduce unnecessary constraints. For example, consider a system with a 16-bit 20-MHz processor, a fast 100-MHz 1-bit serial network interface and several 1-MHz peripheral I/O controllers. If the pure synchronous design methodology is used, the system must be operated at 100 MHz to accommodate the highest clock rate, even though this clock rate is only used in the serial-to-parallel conversion of the serial network interface. It is clear that this system is "overdesigned" for the processor and I/O controllers. The artificial, unnecessarily high clock rate introduces a tighter timing constraint, complicates the design and synthesis process, and increases the hardware complexity. Utilizing separate clock signals can reduce the circuit complexity and simplify the design process.
- *Power consideration.* The dynamic power of a CMOS device is proportional to the switching frequency of transistors, which is correlated to the system clock frequency. An inflated system clock rate will unnecessarily increase the system's power consumption. If we consider the previous system, synchronous design methodology requires the entire system to be operated at 100 MHz. It will consume much more power than three subsystems with clock rates of 100, 20 and 1 MHz.

As discussed in Section 8.2, the synchronous methodology is the fundamental principle in today's digital system development, and most design and analysis schemes are based on

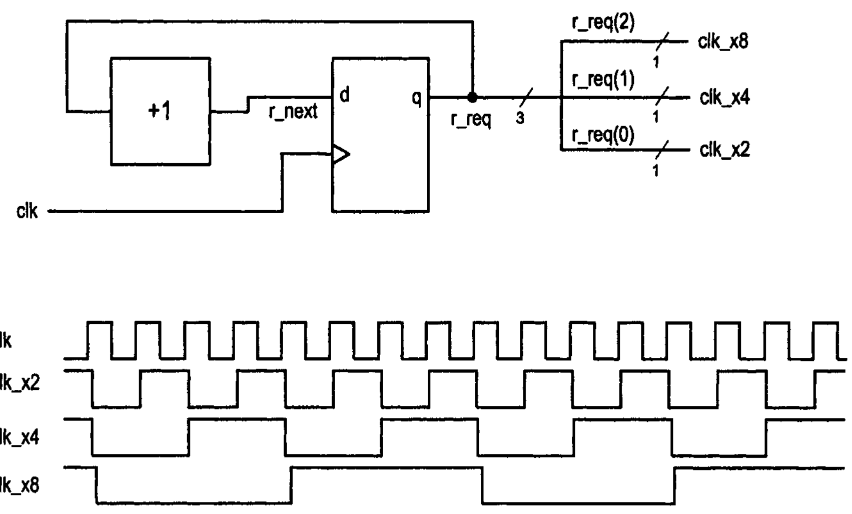


Figure 16.7 Poorly conceived clock divider.

this methodology. Thus, even with multiple clocks, we still want to apply this methodology as much as possible. The basic approach is to divide a system into multiple synchronous subsystems and design a special interface between the subsystems. This allows us to continuously apply the synchronous methodology to design a much larger system.

In a multiple-clock system, the subsystems can use either a *derived* clock signal or an *independent* clock signal. We briefly review the two schemes in the following subsections.

16.3.1 System with derived clock signals

A derived clock signal is a clock signal obtained from a known clock signal. A special clock generation circuit takes the original clock signal, generates new clock signals with different frequencies or phases, and routes them to different subsystems. Each subsystem then utilizes its own clock distribution network to distribute the clock signals to the registers within the subsystem.

In theory, we can apply general RT-level design technique to modify the frequency of a clock signal. For example, we can obtain three lower-frequency clock signals by tapping the output of 3-bit binary counter, as shown in Figure 16.7. There are two problems with this approach. First, because of the clock-to-q delay, there is a skew between the rising edges of original clock signal and the derived clock signals. Second, due to the variation of the clock-to-q delays of the FFs and the unknown wiring delays, it is difficult to determine the exact values of the skews among the three derived clock signals.

To control the skew between the clock signals, the clock generation circuit should be separated from regular logic, and manually analyzed and implemented. Special analog components, such as delay lines and buffers, or even a phase-locked loop (PLL), can be used to minimize the skew.

A system with derived clock signals is subjected to the same setup and hold time constraints. Once the clock skew between the two clock signals is known, we can apply the technique in Section 16.2 to analyze timing. The derivation procedure is similar except that we must take into consideration the difference in the clock periods.

In a multiple-clock system, we use the term *clock domain* to describe use of the clock signal. A clock domain is a block of circuitry in which the FFs are controlled by the same clock signal. Although a derived clock signal has a different clock frequency or phase, its relationship to the original clock signal is known. The design and analysis techniques of synchronous sequential circuit can be modified and applied in such a system. Because of this, we consider that subsystems with derived clock signals are in the same clock domain. Note that these derive clock signals need their own individual clock distribution networks even though they are in the same clock domain.

16.3.2 GALS system

Due to the clock skew of large circuit size or inherent I/O characteristics, it is sometimes difficult or impossible to maintain or find the relationship between the clock signals of subsystems. The clock signals in these subsystems are considered to be independent, and each subsystem constitutes its own clock domain.

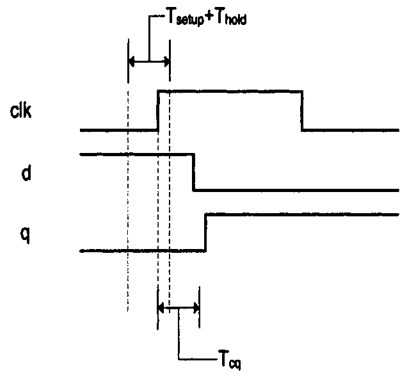
Within a clock domain, the circuit operation is completely synchronous and its design follows the synchronous design methodology. Interface between the two clock domains involves two independent clocks and thus is asynchronous. This configuration is sometimes known as a *globally asynchronous locally synchronous* (*GALS*) system. After we develop a proper asynchronous interface, this scheme allows us to continuously apply the synchronous methodology to design a much larger system.

The major difficulty in designing a GALS system is the interface of clock domains; i.e., how to exchange information and transfer data between two clock domains (known as *domain crossing*). Since the circuit in one domain has no clock information about another domain, a signal may switch at the clock's sampling edge of another domain, which leads to a setup or hold time violation. Recall that one main advantage of the synchronous design methodology is that it provides a systematic way to *avoid a timing violation*. Since a timing violation in the domain crossing is not avoidable, the design must focus on what to do *after a timing violation occurs*.

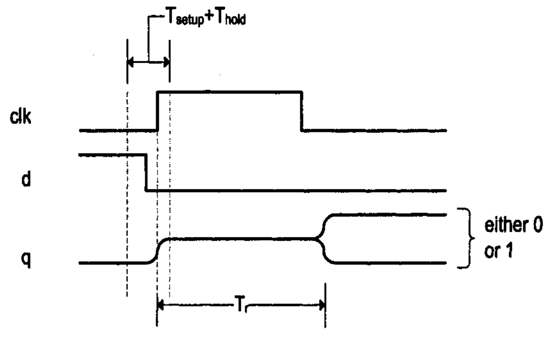
The interface between clock domains is very different from a regular synchronous system or a system with derived clock signals. Its design cannot be automated and usually needs detailed manual analysis. It is more difficult and error-prone. Furthermore, the existence of multiple clock domains affects other processes in the development flow and complicates the static timing analysis, formal verification and test circuit synthesis. Thus, before adding an additional clock domain, we must carefully consider the trade-off between the benefits and potential complexities. In general, it is warranted only for a substantially sized subsystem or a critical high-performance subsystem. The subsequent sections discuss the nature of synchronization failure, the design of a synchronization circuit, and the design and implementation of data transfer protocols.

16.4 METASTABILITY AND SYNCHRONIZATION FAILURE

One fundamental timing constraint of a sequential circuit is the setup and hold times of an FF. It specifies that the input data to an FF must be stable in a decision window around the sampling edge of the clock signal. Consider the basic sequential circuit block diagram shown in Figure 8.5. The input of the register is the next-state logic's output, which is obtained from the register's output and an external input.



(a) Input stable during setup and hold time



(b) Input changing during setup or hold time

Figure 16.8 Timing diagrams of a D FF.

Since the register's output is based on the sampling edge of the clock, we have timing information about this signal. Our timing analysis examines the closed loop formed by the register and next-state logic and ensures that no timing violation will occur. Similar analysis can be performed if the external input signal is generated in the same clock domain. On the other hand, if the external input signal comes from another clock domain, as in a GALS system, the subsystem has no information about the timing relation to its clock, and thus the signal is treated as an asynchronous signal. An asynchronous input signal can change any time, including inside the decision window, and cause a timing violation. The following subsections discuss the characteristics of a timing violation.

16.4.1 Nature of metastability

When an input data signal satisfies the timing constraint, the sampled value will be propagated to the FF's output after the clock-to-q (T_{cq}) delay, as shown in Figure 16.8(a). On the other hand, if the input signal changes during setup or hold time, it violates the timing constraint and the output response is very different. Assume that the input changes from '0' to '1' during the setup and hold time window. One of three scenarios happens:

- The output of the FF becomes '1'.
- The output of the FF becomes '0'.

- The FF enters a *metastable state*, and the output exhibits an in-between voltage value.

The first scenario is the desired result and causes no problem. The second scenario implies that the FF just sampled the previous value. If the input remains unchanged, the correct value will be sampled at the next rising edge. Since we make no assumption about the arrival time of the input signal, there will be no ill effect.

The third scenario is the troublesome one. In normal operation, an FF stays in one of the two stable states, and its output voltage is either high or low. They are interpreted as logic 1 or logic 0 if the positive logic is used. When an FF enters a metastable state, its output voltage is somewhere between the low and the high, and cannot be interpreted as either logic 0 or logic 1. If the output of the FF is used to drive other logic cells, the in-between value may propagate to downstream logic cells and lead the entire digital system into an unknown state.

As its name indicates, a metastable state is not really a stable state. A small noise or disturbance will offset its "balance" and force the FF to enter one of the stable states. In other words, the FF will eventually resolve to a stable state. The time required to reach a stable state is known as the *resolution time*, T_r. The timing diagram is shown in Figure 16.8(b). Theoretical study shows that a bistable device always has a metastable state, and this phenomenon is unavoidable. The only solution is to provide enough time to let the device resolve the situation and reach a stable state.

The resolution time, unfortunately, is not deterministic. It is characterized by a probability distribution function

$$P(T_r) = e^{-\frac{T_r}{\tau}}$$

In this equation, τ is the *decay time constant* and is determined by the electrical characteristics of the FF. A typical value of today's device technology is around a fraction of a nanosecond. The equation indicates the probability that the metastability condition persists beyond T_r after the clock edge. It can be interpreted as the probability that the metastability cannot be resolved within T_r seconds.

16.4.2 Analysis of MTBF(T_r)

Since the timing violation can occur in any asynchronous input, the goal of the design is to confine the metastable condition in an FF and to prevent the in-between value being propagated to the downstream logic. When an FF cannot resolve the metastability condition within the given time, it is known as a *synchronization failure*.

Because of the stochastic nature of the occurrence of a timing violation and resolution time, analysis of the metastable condition is characterized by a statistical average. We use the *average* time interval between two synchronization failures to express the reliability of the design. It is known as *mean time between synchronization failures* (*MTBF*) and is the main quantity used in metastability timing analysis. MTBF depends on many factors. However, in a realistic design scenario, most factors cannot be easily changed, and the only freedom we have is to provide proper resolution time. Thus, MTBF is frequently expressed as a function of T_r, as in MTBF(T_r).

We can derive the MTBF by calculating the average rate of synchronization failures, AF, which is the reciprocal of MTBF. AF is defined as the average number of synchronization failures occurring in a 1-second interval. It is determined by two factors:

- R_{meta}: The average rate at which an FF enters the metastable state.
- $P(T_r)$: The probability that an FF cannot resolve the metastable condition within T_r.

R_{meta} is determined by the formula

$$R_{meta} = w * f_{clk} * f_d$$

In this formula, w is the *susceptible time window*, which is a constant determined by the electrical characteristics of the FF. It can be interpreted as a metastability susceptible time interval associated with the triggering edge of the clock signal. For current device technology, the typical value of w is from few picoseconds to a fraction of a nanosecond. The f_{clk} parameter is the frequency of the clock signal, which is defined as the number of clock cycles per second. During the 1-second interval, there are f_{clk} triggering edges, and thus the $w * f_{clk}$ portion of 1 second is susceptible for the metastability. The f_d parameter is the rate of change in input data, which is defined as the number of input changes per second. We assume that the input data is independent of the clock and that the change can occur at any time. The probability of a single change occurring within a metastability susceptible interval is $w * f_{clk}$. Since there are f_d changes in 1 second, the FF will enter the metastability state $w * f_{clk} * f_d$ times per second, as shown in the equation above.

Once an FF enters the metastable state, it takes a certain amount of time to resolve to a stable state. The discussion in Section 16.4.1 shows that the probability that the FF cannot resolve the metastable condition within the given resolution time of T_r is

$$P(T_r) = e^{-\frac{T_r}{\tau}}$$

In other words, when an FF enters the metastable state, it may resolve the condition within the given resolution time. Only $P(T_r)$ of the events persists over T_r and leads to synchronization failure. Since the FF enters the metastability state R_{meta} times per second on average and only $P(T_r)$ of the entries leads to synchronization failure for the given T_r, the average number of synchronization failures per second is $R_{meta} * P(T_r)$; that is,

$$AF(T_r) = R_{meta} * P(T_r) = w * f_{clk} * f_d * e^{-\frac{T_r}{\tau}}$$

For a given T_r, MTBF(T_r) becomes

$$\text{MTBF}(T_r) = \frac{1}{AF(T_r)} = \frac{e^{\frac{T_r}{\tau}}}{w * f_{clk} * f_d}$$

Note that the f_{clk}, f_d, w and τ parameters are associated with original system specifications and device technology, and revising them can lead to significant design modification. The only freedom we have is to adjust the resolution time (T_r) in the synchronization circuit. That is why we normally express MTBF as a function of T_r, as in MTBF(T_r).

In the MTBF calculation, τ and w depend on the electrical characteristics of the device, and their values can be found in the manufacturer's data sheet. Note that some FPGA manufacturers define the resolution time as the additional time needed after the regular clock-to-q delay (T_{cq}). If we call this time T_{r2}, the relationship between T_r and T_{r2} is

$$T_r = T_{cq} + T_{r2}$$

After simple mathematical manipulation, we can easily convert MTBF(T_{r2}) to MTBF(T_r), and vice versa.

Table 16.1 Sample MTBF(T_r) computation

T_r	**MTBF**
0.0 ns	$4.00 * 10^{-05}$ sec (0.04 msec)
2.5 ns	$5.94 * 10^{-03}$ sec (5.94 msec)
5.0 ns	$8.81 * 10^{-01}$ sec (0.88 sec)
7.5 ns	$1.31 * 10^{+02}$ sec (131 sec)
10.0 ns	$1.94 * 10^{+04}$ sec (5.39 hours)
12.5 ns	$2.88 * 10^{+06}$ sec (3.33 days)
15.0 ns	$4.27 * 10^{+08}$ sec (1.36 years)
17.5 ns	$6.34 * 10^{+10}$ sec ($2.01 * 10^3$ years)
20.0 ns	$9.42 * 10^{+12}$ sec ($2.99 * 10^5$ years)
22.5 ns	$1.40 * 10^{+15}$ sec ($4.43 * 10^7$ years)
25.0 ns	$2.07 * 10^{+17}$ sec ($6.58 * 10^9$ years)
27.5 ns	$3.08 * 10^{+19}$ sec ($9.76 * 10^{11}$ years)
30.0 ns	$4.57 * 10^{+21}$ sec ($1.45 * 10^{14}$ years)
32.5 ns	$6.78 * 10^{+23}$ sec ($2.15 * 10^{16}$ years)
35.0 ns	$1.01 * 10^{+26}$ sec ($3.19 * 10^{18}$ years)

16.4.3 Unique characteristics of MTBF(T_r)

We have examined various timing parameters, such as propagation delay, setup time and hold time. The metastability resolution time is very different. It is not deterministic and not even bounded, and thus must be characterized by a probability distribution function. Note that the resolution time is random in nature, and MTBF, as its name shows, is an average value. When a system has an MTBF value of 1 year, it does not mean that the synchronization failure always happens once a year. It means that the synchronization failure happens once a year *on average*. The actual interval can be 1 month, 6 months, 1 year, 2 years, 5 years and so on. A system may fail in a year regardless of whether its MTBF value is 1 year, 10 years or 1000 years. However, the probability of failure for the system with a 1000-year MTBF is much smaller.

Another observation about the resolution time relates to its highly non-linear characteristics. Note that T_r is in the exponent position of the MTBF(T_r) formula. A small variation over T_r leads to drastic change in the value of MTBF. For example, consider an FF with a w of 0.1 ns and a τ of 0.5 ns and assume that the system clock frequency (f_{clk}) is 50 MHz and the data rate (f_d) is $0.1 f_{clk}$. The resolution time of a synchronizer is normally ranged between a fraction of a clock period to one or two clock periods (discussed in the next section). Table 16.1 lists the MTBF values of T_r from 0 to 35 ns at increments of 2.5 ns. Note that the period of a 50-MHz clock signal is 20 ns. When no resolution time is provided (i.e., $T_r = 0$), the MTBF is an unacceptable 0.04 ms. If we can use a T_r value of half a clock period (i.e., 10 ns), the MTBF becomes about 5 hours. Because of the exponential rate, each extra 2.5 ns can increase the MTBF more than 100 times. When T_r reaches 17.5 ns, the MTBF reaches about 2000 years. If we provide 1.5 times the clock period (i.e., 30 ns), the MTBF becomes about 10^{14} years (for comparison, the age the universe is on the order of 10^{11} years, and the appearance of the human being is on the order of 10^5 years).

This phenomenon is a mixed blessing. On the positive side, while the synchronizing failure cannot be eliminated, we can make the probability of occurrence extremely small.

On the negative side, because of the sensitivity of the resolution time, a small decrease in the resolution time can significantly degrade the value of MTBF. Thus, minor revisions in the system, such as the slight increment of the system clock rate or use of an FF with a slightly larger setup time, may lead to a drastic consequence.

16.5 BASIC SYNCHRONIZER

When an asynchronous input causes a setup or hold time violation, the FF may enter the metastable state and its output exhibits an in-between value. If not blocked, the in-between value will be passed to the next stage and gradually propagated through the entire system.

As its name shows, a *synchronization circuit* (or a *synchronizer*) is to synchronize an asynchronous input with the system clock. As we learned from the previous sections, no circuit can prevent the occurrence of the metastability of a bistable device. The purpose of a synchronization circuit is to stop the propagation of the in-between value and confine the metastability condition within the synchronizer. Since the metastability condition will eventually resolve itself, the task of a synchronizer is just to provide enough time for the FF to reach a stable state.

The following subsections analyze various configurations of a synchronizer and their MTBFs. In our examples, we assume that the circuit utilizes the FF of Section 16.4.3, and has same clock frequency and data rate; i.e., w = 0.1 ns, τ = 0.5 ns, f_{clk} = 50 MHz and $f_d = 0.1 f_{clk}$.

16.5.1 The danger of no synchronizer

We first consider a sequential circuit that has no synchronizer for its asynchronous input, as shown in Figure 16.9(a). If the asynchronous signal causes a timing violation, the system register may enter the metastable state, and the in-between value will be propagated to the next-state logic circuit. We can analyze how frequently the system enters the metastable state using the previous MTBF formula. Since there is no synchronizer, no resolution time is provided (i.e., $T_r = 0$). Substituting this value into the formula, we have MTBF(0) = 0.04 ms. This failure rate is clearly unacceptable.

16.5.2 One-FF synchronizer and its deficiency

The first design of a synchronizer is to use a single D FF, as shown in Figure 16.9(b). Let T_c, T_{setup} and T_{comb} be the clock period of the system, the setup time of the FF and the propagation delay of the combinational circuit respectively. Consider the path from the synchronizer D FF to the system D FF. The synchronizer provides one clock period for the out_sync signal to resolve, propagate through the combinational logic and satisfy the setup time constraint of the system D FF. The required time for the latter two is $T_{comb} + T_{setup}$, and the remaining balance can be used to resolve the metastability condition, which is

$$T_r = T_c - (T_{comb} + T_{setup})$$

Assume that T_{setup} of the system register is 2.5 ns. The resolution time of this circuit becomes

$$T_r = 20 - (T_{comb} + 2.5) = 17.5 - T_{comb}$$

T_r and MTBF depend on T_{comb}, the propagation delay of the combinational circuit. For a simple combination circuit, the T_r will be relatively large. For example, if T_{comb} is 1 ns,

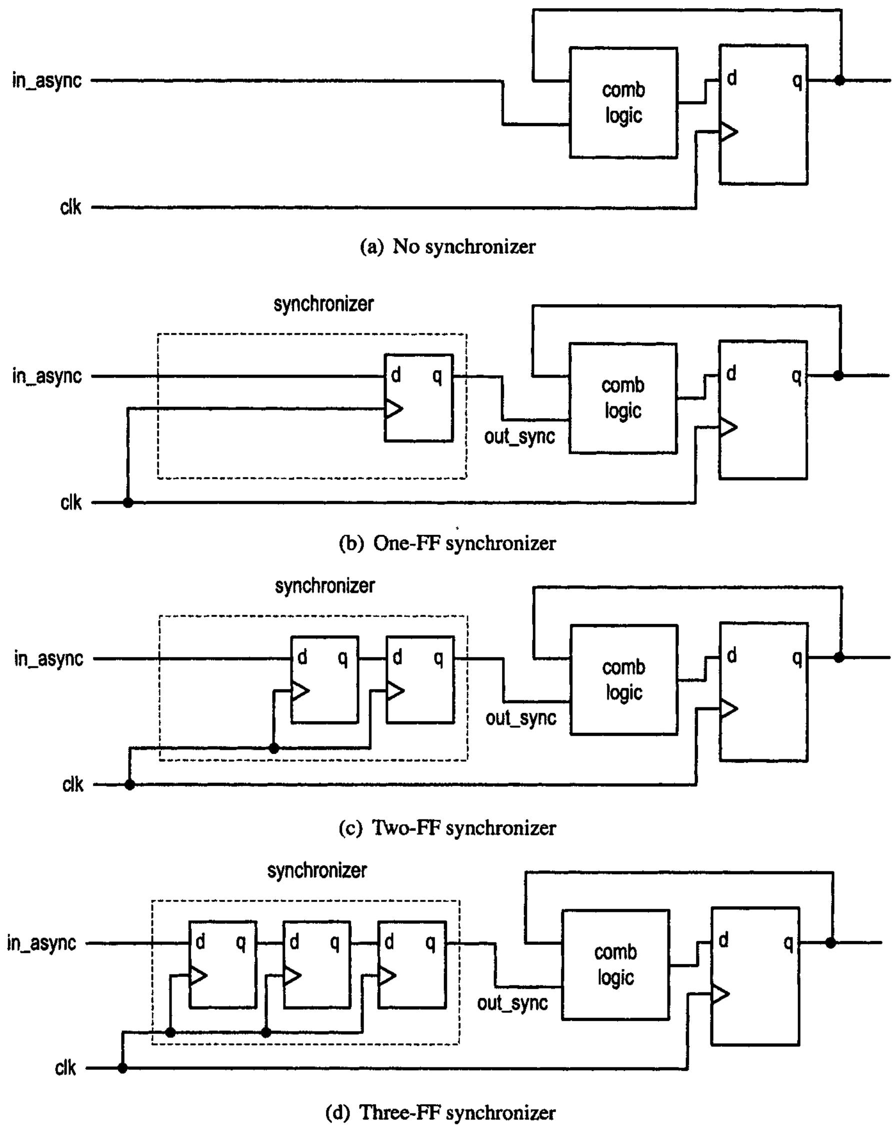


Figure 16.9 Synchronizers.

T_r becomes 16.5 ns and MTBF(16.5 ns) is about 272 years. On the other hand, a complex combination circuit can drastically reduce the MTBF value. If T_{comb} is 12.5 ns, T_r becomes 5 ns and MTBF(5 ns) is dropped to about 0.88 second.

As discussed earlier, MTBF is extremely sensitive to T_r, and a small variation leads to a huge swing in MTBF value. The value of T_{comb} depends on the logic function of the combinational circuit, device technology as well as synthesis and placement and routing process, and thus cannot be determined in advance. A minor modification in the combinational logic, the synthesis process or the placement and routing process can lead to a significant reduction in MTBF and make the system susceptible to synchronization failure. Therefore, this is not a reliable design. A better alternative is to use two D FFs for the synchronizer.

16.5.3 Two-FF synchronizer

The previous analysis shows that a maximal resolution time can be obtained if T_{comb} is 0. Since the function of the combinational logic is defined by the original system, we cannot modify it arbitrarily. Instead, we can insert another D FF to form a two-FF synchronizer, as shown in Figure 16.9(c). The resolution time provided by the two FFs inside the synchronizer is

$$T_r = T_c - T_{setup}$$

If T_{setup} is 2.5 ns, the resolution time becomes

$$T_r = 20 - 2.5 = 17.5 \text{ ns}$$

The MTBF(17.5 ns) is about 3000 years. In addition to providing more resolution time, this design is also more robust since no logic function or synthesis is involved. The only uncertain factor in this design is the wiring delay, which can be substantial if the two D FFs are located far apart. To minimize this delay, the two D FFs must be placed as close as possible. In physical design, we may need to manually perform the placement and routing for the synchronizer.

The VHDL code for the synchronizer is straightforward, following the block diagram of Figure 16.9(c). The code is shown in Listing 16.1.

Listing 16.1 Two-FF synchronizer

```
library ieee;
use ieee.std_logic_1164.all;
entity synchronizer is
   port(
      clk, reset: in std_logic;
      in_async: in std_logic;
      out_sync: out std_logic
   );
end synchronizer;

architecture two_ff_arch of synchronizer is
   signal meta_reg, sync_reg: std_logic;
   signal meta_next, sync_next: std_logic;
begin
   -- two D FFs
   process(clk,reset)
   begin
      if (reset='1') then
         meta_reg <= '0';
         sync_reg <= '0';
      elsif (clk'event and clk='1') then
         meta_reg <= meta_next;
         sync_reg <= sync_next;
      end if;
   end process;
   -- next-state logic
   meta_next <= in_async;
   sync_next <= meta_reg;
   -- output
   out_sync <= sync_reg;
end two_ff_arch;
```

Because of its simplicity and robustness, the two-FF configuration is the most widely used synchronizer. It is satisfactory in most applications. However, the regular D FF occasionally may not be able to provide sufficient T_r. For example, if we increase the system clock by one-third to 66.7 MHz, the clock period is reduced to 15 ns and T_r becomes 12.5 ns. The MTBF is reduced to 3.33 days. To overcome this problem, many ASIC technologies have a special *metastability-hardened* D FF cell in their libraries. The functionality of this D FF is identical to that of a regular D FF, but its w, τ and T_{setup} are made smaller to increase MTBF. We can use component instantiation in VHDL code to instantiate this type of D FF cell in a synchronizer. Due to its internal implementation, the circuit size of a metastability-hardened D FF cell is several times larger than that of a regular D FF cell and thus should not be used in regular sequential circuits.

16.5.4 Three-FF synchronizer

If the device technology does not provide a metastability-hardened D FF cell, we can increase the resolution time by cascading more D FF cells or artificially enlarging the clock period of the synchronizer. The three-FF synchronizer is shown in Figure 16.9(d). An extra D FF is cascaded with a two-FF synchronizer. The idea behind this design is to use the extra D FF to provide an additional opportunity to resolve the metastability condition.

We can follow the procedure in Section 16.4.2 to calculate the MTBF of this circuit. Recall that R_{meta}, the average rate at which the first D FF enters a metastable state, is

$$R_{meta} = w * f_{clk} * f_d$$

Once the FF enters the metastable state, it has a time interval of $T_c - T_{setup}$ to resolve the situation. The probability that the metastability condition persists beyond the current clock cycle is

$$P1 = e^{-\frac{T_c - T_{setup}}{\tau}}$$

If this situation happens, the metastability condition is sampled and passed to the second FF. The second FF, again, has a time interval of $T_c - T_{setup}$ to resolve the situation, and the probability that the metastability condition persists beyond this clock cycle is

$$P2 = e^{-\frac{T_c - T_{setup}}{\tau}}$$

The MTBF of this circuit becomes

$$\text{MTBF} = \frac{1}{R_{meta} * P1 * P2} = \frac{e^{\frac{2(T_c - T_{setup})}{\tau}}}{w * f_{clk} * f_d}$$

If we compare this equation to the two-FF synchronizer, which is

$$\text{MTBF} = \frac{1}{R_{meta} * P1} = \frac{e^{\frac{(T_c - T_{setup})}{\tau}}}{w * f_{clk} * f_d}$$

We can interpret that the three-FF synchronizer increases the resolution time from $T_c - T_{setup}$ to $2(T_c - T_{setup})$.

Since this term is in the exponent of the equation, its impact is very significant. If T_c is 20 ns and T_{setup} is 2.5 ns, the resolution time increases from 17.5 ns to 35 ns, and the MTBF increases from 2000 thousand years to 10^{18} years. If T_c is 15 ns, the resolution time increases from 12.5 ns to 25 ns and the MTBF increases from 3 days to about 6 billion years, which is a pretty safe number.

The disadvantage of the three-buffer synchronizer is the delay for the input signal. The extra D FF increases the delay from two clock cycles to three clock cycles. When possible, we should use a metastability-hardened D FF cell rather than using an additional D FF.

We can cascade more D FFs to increase the MTBF. However, because of the effect of the exponential decay, this is seldom needed in reality.

16.5.5 Proper use of a synchronizer

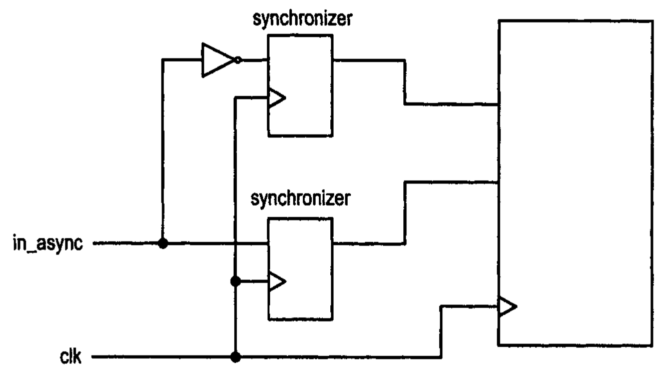
The function of a synchronizer is to provide a non-metastable output value. We must use it properly to obtain a reliable synchronized result. Good design practices can help us to achieve this goal and avoid subtle errors:

- Use a glitch-free signal for synchronization.
- Synchronize a signal in a single place.
- Avoid synchronizing multiple "related" signals.
- Reanalyze the synchronizer after each design change.

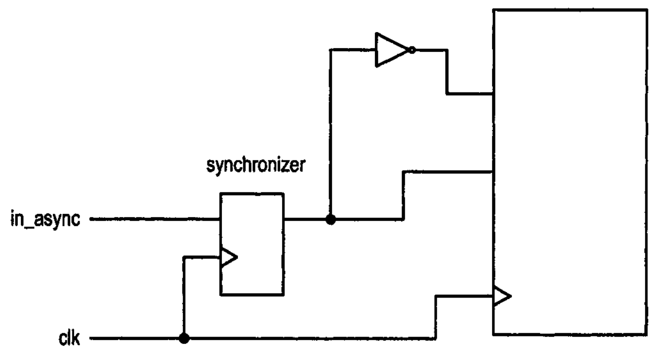
These practices are discussed in the following paragraphs.

Use a glitch-free signal for synchronization The asynchronous input signal normally comes from another clock domain. Since the synchronizer has no knowledge about the clock signal in another domain, it can sample the asynchronous input any time. If a glitch exists in the input signal, it may be sampled and synchronized incorrectly as a legitimate value. It is important to pass a glitch-free signal for synchronization. This can be achieved by adding an output buffer when the signal is generated.

Synchronize a signal in a single place The function of a synchronizer is to generate a stable output value. The synchronizer, however, cannot guarantee which value will be reached. For example, if a timing violation occurs when the input changes from '0' to '1', the synchronized input value can be '0' or '1' at the current sampling clock edge. Assume that the input signal does not change. It will be sampled again at the next rising edge of the clock and a stable '1' will be obtained. This implies that the arrival time of a synchronized asynchronous input signal may exhibit a random one-clock delay. We must take the random delay into consideration when using a synchronizer.



(a) Synchronizing a signal in two places



(b) Synchronizing a signal in one place

Figure 16.10 Synchronization at multiple places.

An asynchronous input signal may be used in multiple places in a clock domain. It should be synchronized in a single entry point. An example of a poor design is shown in Figure 16.10(a), in which the `in_async` signal and its derivative are synchronized by two individual synchronizers. The potentially random one-clock delay may introduce inconsistent values to the system and lead to incorrect operation. A better alternative is shown in Figure 16.10(b). The signal is synchronized by a single synchronizer, and the system is always fed with the same value.

Avoid synchronizing related signals A similar issue is to synchronize related signals. *Related signals* means that a group of signals are combined to represent a command, state and so on. For example, we may use two signals to represents four possible actions. Because of the random one-clock delay, synchronizing related signals may to lead to uncertain results. For example, consider that two related signals changes from "00" to "11". If the two signals switch at about the same time and both transitions cause timing violations, the resolved results can be "00", "01", "10" or "11" for one clock cycle. Although the signal will eventually be settled to "11" in the next clock cycle, the "01" and "10" conditions may exist for one clock cycle. This may cause a serious problem for some applications.

There are two ways to correct the problem. The first is to apply special coding patterns, such as Gray code, to ensure that only one bit changes during the transition. One example is given in Section 16.9.1. A better, more systematic alternative is to bundle all signals and pass them as a single data item. The data transfer between two clock domains is discussed in Section 16.8.

Reanalyze the synchronizer after each design change MTBF is extremely sensitive to the available resolution time, and a small variation can lead to drastic change. For example, consider the two-FF synchronizer discussed in Section 16.5.3. If the original system is running at 50 MHz, the MTBF is about 3000 years. Assume that we upgrade the design using faster functional units and the new system can run at 66.7 MHz, about 33% faster. Since the same device technology is used for the D FFs, w and τ remain unchanged. The MTBF is reduced to a mere 3 days, which is only 0.0002% of the original value. The example demonstrates the sensitivity of the synchronizing circuit. It is good practice to examine the synchronizer after each design modification.

16.6 SINGLE ENABLE SIGNAL CROSSING CLOCK DOMAINS

In a GALS system, clock domains are driven by independent clock signals. The clock frequencies and data processing rates of these domains may not be identical. A subsystem can communicate with another subsystem whose clock frequency is 10 times faster or 10 times slower. The function of a synchronizer is to prevent the subsystems from entering the metastable state. Additional control schemes are needed to coordinate the information exchange between the two clock domains. We show how to propagate an enable pulse signal from one clock domain to another clock domain in this section and Section 16.7 and discuss the data transfer in Sections 16.8 and 16.9.

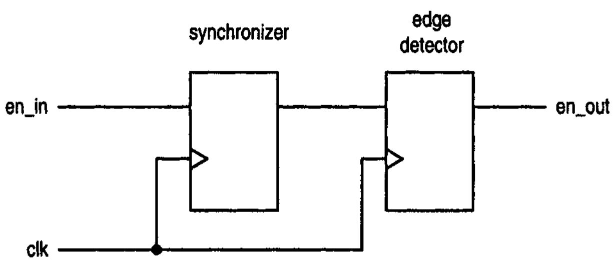
16.6.1 Edge detection scheme

A digital system frequently includes a control signal in the form of an enable pulse, which activates the desired action for a single time. The enable signal of a counter and the start signal of a sequential multiplier are signals of this type. An enable pulse should be sampled by exactly one clock edge. A longer duration may cause errors. For example, a counter may count twice for a single event or a multiplier may load the incorrect operands.

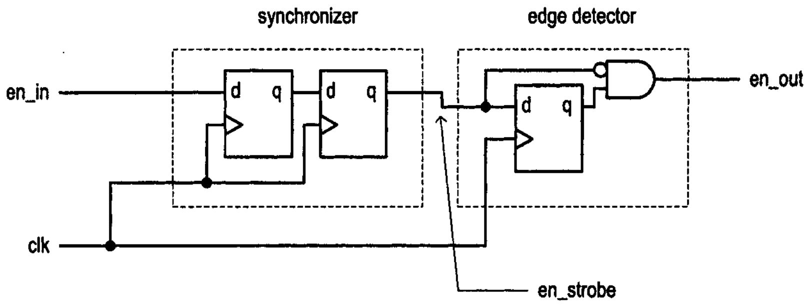
While using an enable pulse between two synchronous subsystems is straightforward, it is much harder to pass the pulse crossing the clock domains. We must consider the synchronization and the difference in clock rates. The following subsections discuss several ad hoc edge detection schemes to regenerate an enable pulse from a slow or a fast clock domain. A more robust scheme that involves feedback signal is discussed in the next section.

Wide enable signal If an enable pulse is generated from a slow clock domain, its duration may last for several clock cycles in the current clock domain, and the signal appears as a very wide pulse. A rising-edge detection circuit is needed to regenerate a shorter, synchronized enable pulse in the current clock domain. The block diagram is shown in Figure 16.11(a), which includes a synchronizer and an edge detection circuit.

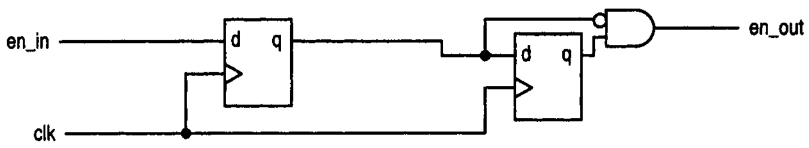
The rising-edge detection circuit can be designed by using an FSM or direct implementation, as discussed in Section 10.4.1. We use the implementation shown in Figure 10.19 of Section 10.8.1, and its VHDL code is repeated in Listing 16.2.



(a) Block diagram



(b) Correct circuit diagram



(c) Incorrect implementation

Figure 16.11 Regeneration of a wide enable signal.

Listing 16.2 Rising-edge detection circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity rising_edge_detector is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      pulse: out std_logic
   );
end rising_edge_detector;

architecture direct_arch of rising_edge_detector is
   signal delay_reg: std_logic;
begin
   -- delay register
   process(clk,reset)
   begin
      if (reset='1') then
         delay_reg <= '0';
      elsif (clk'event and clk='1') then
         delay_reg <= strobe;
      end if;
   end process;
   -- decoding logic
   pulse <= (not delay_reg) and strobe;
end direct_arch;
```

After substituting the gate-level implementation in the blocks, we can obtain a more detailed circuit diagram, as shown in Figure 16.11(b).

The VHDL code for the complete enable pulse regeneration circuit is shown in Listing 16.3. We intentionally use the component instantiation and create two component instances in the top-level description to highlight the use of a synchronizer and to differentiate it from a regular sequential circuit. After each design change, the synchronizer instance must be reexamined and, if needed, replaced, to ensure the proper MTBF.

Listing 16.3 Enable pulse regenerator for a wide enable signal

```
library ieee;
use ieee.std_logic_1164.all;
entity sync_en_pulse is
   port(
      clk, reset: in std_logic;
      en_in: in std_logic;
      en_out: out std_logic
   );
end sync_en_pulse;

architecture slow_en_arch of sync_en_pulse is
   component synchronizer
      port(
         clk, reset: in std_logic;
         in_async: in std_logic;
         out_sync: out std_logic
      );
   end component;
   component rising_edge_detector
      port(
         clk, reset: in std_logic;
         strobe: in std_logic;
         pulse: out std_logic
      );
   end component;
   signal en_strobe: std_logic;
begin
   sync: synchronizer
      port map (clk=>clk, reset=>reset, in_async=>en_in,
                out_sync=>en_strobe);
   edge_detect: rising_edge_detector
      port map (clk=>clk, reset=>reset, strobe=>en_strobe,
                pulse=>en_out);
end slow_en_arch;
```

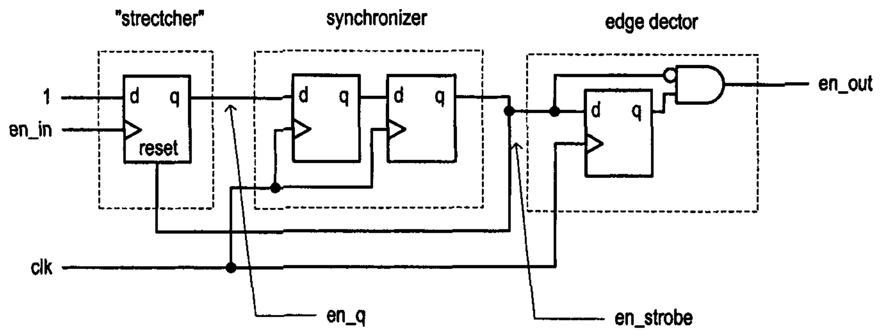


Figure 16.12 Regeneration of a narrow enable signal.

We may be tempted to use the second D FF of the synchronizer to function as the edge detection circuit to save a D FF and to reduce the propagation delay, as shown in Figure 16.11(c). This is a poor design since the unresolved signal may leak through the and cell and propagate to the downstream logic.

Narrow enable signal Handling an enable pulse from a fast clock domain is more difficult. For example, if the pulse is generated from a domain whose clock frequency is eight times faster than the frequency of the current clock domain, the duration of the enable pulse is only one-eighth of the period of the current clock signal. The sampling edge of the D FF of the synchronizer is likely to miss the narrow pulse.

Since the signal cannot be sampled by the clock edge, no synchronous design method can solve this problem. We must turn to ad hoc techniques to "stretch" the pulse until it is sampled by the current clock. One possible design is shown in Figure 16.12. In this design, the enable pulse is used as the clock for the stretcher D FF. When a pulse arrives, the stretcher D FF is loaded with '1'. The output of the D FF is then passed to the synchronizer. After the pulse is synchronized, the asserted synchronizer output clears the first D FF via the asynchronous reset. Due to the random one-clock delay of the synchronizer, the duration of the synchronized output can be one or two clock periods, and thus an edge detection circuit is needed to ensure correct operation. Because the first D FF is driven by a different clock signal, it should be excluded for the regular timing analysis and testing circuit. The VHDL code of the revised architecture body is shown in Listing 16.4.

Listing 16.4 Enable pulse regenerator for a narrow enable signal

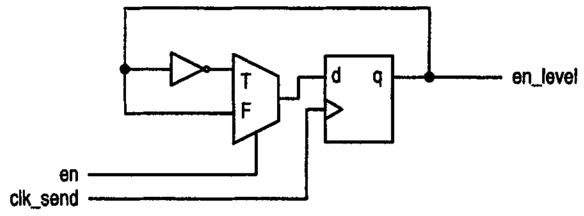
```
architecture fast_en_arch of sync_en_pulse is
   component synchronizer
      port(
         clk, reset: in std_logic;
         in_async: in std_logic;
         out_sync: out std_logic
      );
   end component;
   component rising_edge_detector
      port(
         clk, reset: in std_logic;
         strobe: in std_logic;
         pulse: out std_logic
      );
   end component;
   signal en_strobe: std_logic;
   signal en_q: std_logic;
begin
   -- ad hoc stretcher
   process(en_in,en_strobe)
   begin
      if (en_strobe='1') then
         en_q <= '0';
      elsif (en_in'event and en_in='1') then
         en_q <= '1';
      end if;
   end process;
   -- slow enable pulse regenerator
   sync: synchronizer
      port map (clk=>clk, reset=>reset, in_async=>en_q,
                out_sync=>en_strobe);
   edge_detect: rising_edge_detector
      port map (clk=>clk, reset=>reset, strobe=>en_strobe,
                pulse=>en_out);
end fast_en_arch;
```

Since the function of the first D FF depends only on the rising edge, not on the duration, of the incoming pulse, this scheme can be applied to a wide enable pulse as well. Note that the incoming enable pulse must be glitch-free to prevent false triggering.

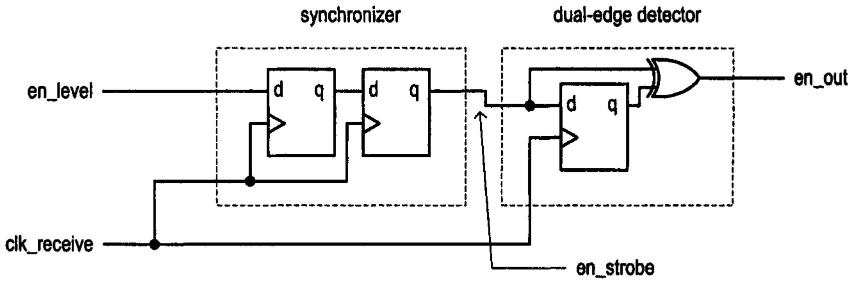
16.6.2 Level-alternation scheme

An alternative to the ad hoc pulse-stretching circuit is to slightly modify the interface between the two clock domains and use the alternation of the output level to carry the information. In this scheme, the sending subsystem toggles the output value when an enable pulse is generated and thus embeds the pulse information into the signal transition edges. The block diagram is shown in Figure 16.13(a). The circuit is a T FF, which toggles its output after each time the en signal is asserted. When an enable pulse arrives, the en_level signal switches state, as shown in the top and middle parts of the timing diagram in Figure 16.13(c). The corresponding VHDL segment is

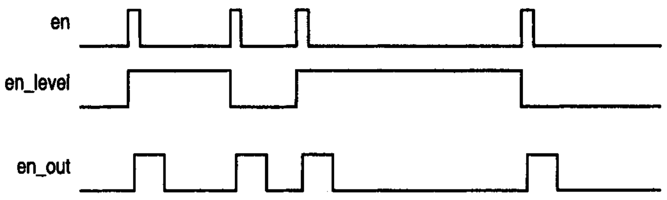
```
. . .
-- D FF
process(clk, reset)
begin
   if (reset='1') then
      t_next <= '0';
   elsif (clk'event and clk='1') then
      t_reg <=  t_next;
   end if;
end process;
-- next-state logic
t_next <= not (t_reg) when en='1' else
          t_reg;
-- output logic
en_level <= t_reg
. . .
```



(a) Block diagram of the sending subsystem



(b) Block diagram of the receiving subsystem



(c) Simplified timing diagram

Figure 16.13 Pulse regeneration with level alternation.

In the domain that receives the enable pulse, it needs a synchronizer and a dual-edge detection circuit that can detect both the rising and falling edges of an input signal. The edge detection circuit senses the change in signal level and converts it back to a single one-clock-period pulse. We can derive the dual-edge detection circuit by using an FSM or direct implementation. One possible direct implementation is to perform an xor operation over the current input value and the previous input value stored in a D FF, as shown in Figure16.13(b). The output waveform of the regenerator is demonstrated in the bottom part of the timing diagram in Figure 16.13(c). For clarity, the synchronizer delay is not included in the diagram.

The VHDL codes for the dual-edge detection circuit and the architecture body of the revised enable pulse regeneration circuit are shown in Listings 16.5 and 16.6 respectively. Note that the new `dual_edge_dector` component is used in the architecture body.

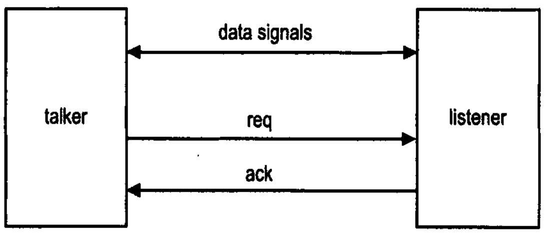
Listing 16.5 Dual-edge detection circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity dual_edge_detector is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      pulse: out std_logic
   );
end dual_edge_detector;

architecture direct_arch of dual_edge_detector is
  signal delay_reg: std_logic;
begin
   -- delay register
   process(clk,reset)
   begin
      if (reset='1') then
         delay_reg <= '0';
      elsif (clk'event and clk='1') then
         delay_reg <= strobe;
      end if;
   end process;
   -- decoding logic
   pulse <= delay_reg xor strobe;
end direct_arch;
```

Listing 16.6 Enable pulse regenerator using the level-alternation scheme

```
architecture level_arch of sync_en_pulse is
   component synchronizer
      port(
         clk, reset: in std_logic;
         in_async: in std_logic;
         out_sync: out std_logic
      );
   end component;
   component dual_edge_detector
      port(
         clk, reset: in std_logic;
         strobe: in std_logic;
         pulse: out std_logic
      );
   end component;
   signal en_strobe: std_logic;
begin
   sync: synchronizer
      port map (clk=>clk, reset=>reset, in_async=>en_in,
                out_sync=>en_strobe);
   edge_detect: dual_edge_detector
      port map (clk=>clk, reset=>reset, strobe=>en_strobe,
                pulse=>en_out);
end level_arch;
```



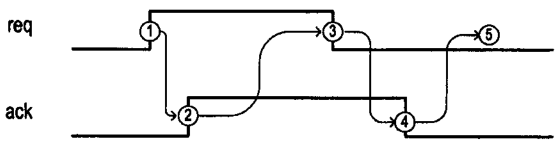


Figure 16.14 Basic conceptual and timing diagrams of the four-phase handshaking protocol.

16.7 HANDSHAKING PROTOCOL

While the pulse regeneration schemes of Section 16.6 can handle an enable signal with different widths, they cannot control the *rate* at which the enable pulses are generated. For example, consider a sending subsystem with a clock frequency that is eight times faster than that of a receiving subsystem. The previous schemes can regenerate the enable pulse in the receiving subsystem even when the input's width is only one-eighth that of the clock period. However, if the enable pulse is generated every four clock cycles, the rate is too fast for the receiving subsystem to process, and some pulses will be lost when crossing the domains. In order to function properly, the sending subsystem needs some knowledge of the receiving subsystem and issues the enable pulse accordingly.

To develop a more robust scheme, we must utilize a feedback signal from the receiving subsystem to communicate its status and establish a rule, which is known as a *protocol*, between the two subsystems. The following subsections discuss a four-phase and a two-phase handshaking protocols. While these protocols can be used to regulate the rate of the arriving enable pulses, their major applications are associated with the data transfer between two clock domains. This subject is discussed in the next section.

16.7.1 Four-phase handshaking protocol

The most commonly used scheme to coordinate operations between two clock domains is the four-phase handshaking protocol. This protocol makes no assumptions about the relative data processing rates between the clock domains and thus can accommodate a wide range of applications. In this protocol, the two subsystems are designated as the *talker* and the *listener* respectively. The talker and the listener exchange information via the `req` signal, which is the request signal from the talker to the listener, and the `ack` signal, which is the acknowledge signal from the listener to the talker. The simplified block diagram is shown in Figure 16.14(a).

The basic operation sequence (i.e., the handshaking procedure) of the four-phase handshaking protocol is illustrated in Figure 16.14(b). It consists of the following steps:

1. The talker activates the `req` signal.

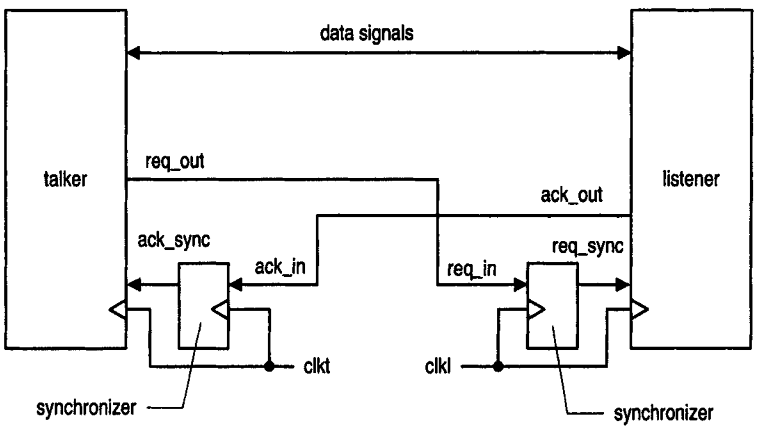


Figure 16.15 Handshaking system with synchronizers.

2. When the listener detects activation of the `req` signal, it activates the `ack` signal to inform the talker.
3. When the talker senses activation of the `ack` signal, it deactivates the `req` signal.
4. After the listener detects deactivation of the `req` signal, it deactivates the `ack` signal accordingly.
5. Once the talker senses deactivation of the `ack` signal, it returns to the initial state. The talker can issue a new request if needed.

In this protocol, the listener provides feedback information via the `ack` signal to let the talker know that a change is detected in the `req` signal, and the talker can respond accordingly. Note that there is no assumption about the operation speed of the listener and the talker. The talker must keep the `req` signal asserted until the `ack` signal is activated. The talker does not need to make any assumptions about the operation speed or the clock rate and can send a signal to a subsystem with unknown characteristics.

Note that we can combine the talker and the listener and treat them as a single system. The values of the `req` and `ack` signals define the "system state." When the `req` and `ack` signals are "00", the system is in the idle or initial state. During the handshaking process, they change to "10", "11" and "01" and eventually return to "00", the original state. We call the protocol *four-phase handshaking* because the sequence progresses through four distinctive states.

Since the `req` and `ack` signals cross the clock domains, two synchronizers are needed in the actual implementation. The more detailed block diagram of the handshaking scheme is shown in Figure 16.15. In the actual implementation, we use the `_in`, `_out` and `_sync` suffixes to indicate that the corresponding signal is an asynchronous input signal, output signal and synchronized input signal respectively.

The protocol can be implemented by two separate FSMs, one for the talker and one for the listener. Their ASM charts are shown in Figure 16.16. We assume that the talker FSM also has an input command, `start`, and an output status, `ready`. The FSM initializes the handshaking operation when the `start` signal is activated and asserts the `ready` signal when it is in the `idle` state. When the sending subsystem wants to issue an enable pulse across the clock domain, it checks the `ready` signal to ensure that the talker FSM is idle and then activates the `start` signal for one clock cycle. After the talker FSM senses the

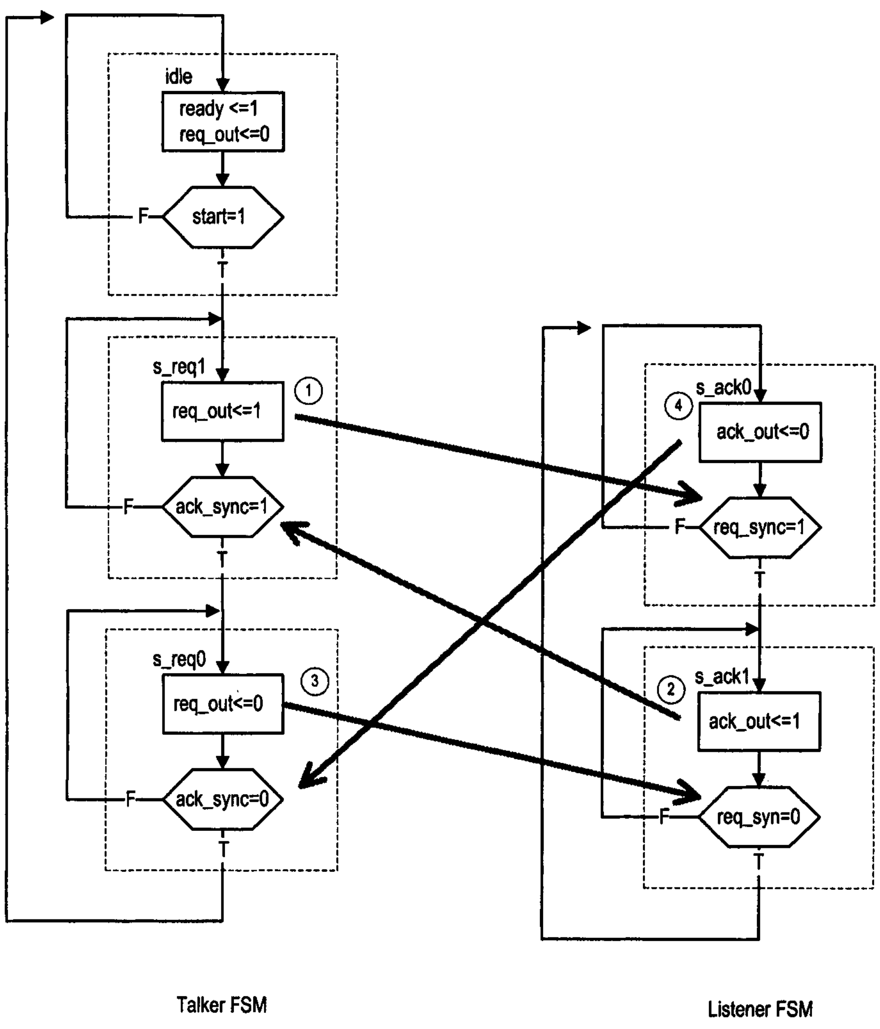


Figure 16.16 ASM charts of the talker and listener of the four-phase handshaking protocol.

start signal, it moves to the s_req1 state, in which the req_out signal is activated. The FSM then stays in the s_req1 state until activation of the acknowledge signal, ack_sync. It then moves to the s_req0 state and deactivates the req_out signal. The FSM returns to the idle state after it senses deactivation of the ack_sync signal.

The listener FSM is similar to the talker FSM except that it contains no start signal and thus can only respond to the talker FSM.

Because the ack_out and req_out signals are to be synchronized by a different clock domain, they must be glitch-free. This can be achieved by adding proper output buffers. Since they are designed as Moore outputs in the FSMs, we use the look-ahead output buffer scheme discussed in Section 10.7.2. The VHDL codes of the two FSMs are shown in Listings 16.7 and 16.8 respectively.

Listing 16.7 Talker FSM of the four-phase handshaking protocol

```
library ieee;
use ieee.std_logic_1164.all;
entity talker_fsm is
   port(
      clk, reset: in std_logic;
      start, ack_sync: in std_logic;
      ready: out std_logic;
      req_out: out std_logic
   );
end talker_fsm;

architecture arch of talker_fsm is
   type t_state_type is (idle, s_req1, s_req0);
   signal state_reg, state_next: t_state_type;
   signal req_buf_reg, req_buf_next: std_logic;
begin
   -- state register and output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         req_buf_reg <='0';
     elsif (clk'event and clk='1') then
        state_reg <= state_next;
        req_buf_reg <=req_buf_next;
     end if;
   end process;
   -- next-state logic
   process(state_reg,start,ack_sync)
   begin
      ready <='0';
      state_next <= state_reg;
      case state_reg is
         when idle =>
            if start='1' then
               state_next <= s_req1;
            end if;
            ready <= '1';
         when s_req1 =>
```

```
            if ack_sync='1' then
               state_next <= s_req0;
            end if;
         when s_req0 =>
            if ack_sync='0' then
                state_next <= idle;
            end if;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      case state_next is
         when idle =>
            req_buf_next <= '0';
         when s_req1 =>
            req_buf_next <= '1';
         when s_req0 =>
            req_buf_next <= '0';
      end case;
   end process;
   req_out<= req_buf_reg;
end arch;
```

Listing 16.8 Listener FSM of the four-phase handshaking protocol

```
library ieee;
use ieee.std_logic_1164.all;
entity listener_fsm is
   port(
      clk, reset: in std_logic;
      req_sync: in std_logic;
      ack_out: out std_logic
   );
end listener_fsm;

architecture arch of listener_fsm is
   type l_state_type is (s_ack0, s_ack1);
   signal state_reg, state_next: l_state_type;
   signal ack_buf_reg, ack_buf_next: std_logic;
begin
   -- state register and output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s_ack0;
         ack_buf_reg <='0';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         ack_buf_reg <= ack_buf_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,req_sync)
   begin
      state_next <= state_reg;
      case state_reg is
         when s_ack0 =>
            if req_sync='1' then
               state_next <= s_ack1;
            end if;
         when s_ack1 =>
            if req_sync='0' then
               state_next <= s_ack0;
            end if;
         end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      case state_next is
         when s_ack0 =>
            ack_buf_next <= '0';
         when s_ack1 =>
            ack_buf_next <= '1';
         end case;
   end process;
   ack_out<= ack_buf_reg;
end arch;
```

To complete the design, synchronizers are needed for the input signals. Again, to emphasize the unique characteristics of the synchronization circuits, we separate them from the regular sequential circuits and instantiate the synchronizers in the top level. The VHDL codes of the complete design follow the block diagram in Figure 16.15 and are shown in Listings 16.9 and 16.10 respectively.

Listing 16.9 Talker of the four-phase handshaking protocol

```
library ieee;
use ieee.std_logic_1164.all;
entity talker_str is
   port(
      clkt: in std_logic;
      resett: in std_logic;
      ack_in: in std_logic;
      start: in std_logic;
      ready: out std_logic;
      req_out: out std_logic
   );
end talker_str;

architecture str_arch of talker_str is
   signal ack_sync: std_logic;
   component synchronizer
      port(
         clk: in std_logic;
         in_async: in std_logic;
         reset: in std_logic;
         out_sync: out std_logic
      );
   end component;
   component talker_fsm
      port(
         ack_sync: in std_logic;
         clk: in std_logic;
         reset: in std_logic;
         start: in std_logic;
         ready: out std_logic;
         req_out: out std_logic
      );
   end component;
begin
   sync_unit: synchronizer
      port map (clk=>clkt, reset=>resett, in_async=>ack_in,
                out_sync=>ack_sync);
   fsm_unit: talker_fsm
      port map (clk=>clkt, reset=>resett, start=>start,
                ack_sync=>ack_sync, ready=>ready,
                req_out=>req_out);
end str_arch;
```

Listing 16.10 Listener of the four-phase handshaking protocol

```
library ieee;
use ieee.std_logic_1164.all;
entity listener_str is
   port(
      clkl: in std_logic;
      resetl: in std_logic;
      req_in: in std_logic;
      ack_out: out std_logic
   );
end listener_str;

architecture str_arch of listener_str is
   signal req_sync: std_logic;
   component listener_fsm
      port (
         clk: in std_logic;
         req_sync: in std_logic;
         reset: in std_logic;
         ack_out: out std_logic
      );
   end component;
   component synchronizer
      port (
         clk: in std_logic;
         in_async: in std_logic;
         reset: in std_logic;
         out_sync: out std_logic
```

```
        );
    end component;
begin
    sync_unit: synchronizer
       port map (clk=>clkl, reset=>resetl, in_async=>req_in,
                 out_sync=> req_sync);
    fsm_unit: listener_fsm
       port map (clk=>clkl, reset=>resetl, req_sync=>req_sync,
                 ack_out => ack_out);
end str_arch;
```

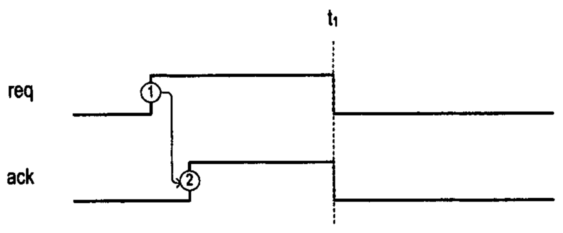
We can use this protocol to pass an enable pulse across the clock domain by connecting the en signal to the start signal of the talker FSM. When an enable pulse arrives, the talker initiates the handshaking operation. When the listener detects the activation edge of the req_in signal, it can also generate an output pulse, which corresponds to the regenerated enable pulse in the new clock domain. Since the sending subsystem cannot generate another enable pulse until the handshaking operation is completed, the sending subsystem will not overrun the the receiving subsystem.

At first glance, the four phases may appear to be somewhat redundant. We may be tempted to discard the second half of the handshaking to simplify the FSMs and let the talker and listener return to the initial state automatically. Let us consider what happens if this is done. Assume that the talker and the listener deactivate the req and ack signals automatically after the system reaches the "11" phase. There will be no problem if the deactivations are done simultaneously, as shown in the timing diagram of Figure 16.17(a). However, since the two subsystems are driven by different clocks, this is hardly possible. If the talker is much slower, the listener may be fooled into thinking that the asserted req signal is the initiation of a new request, as shown in the timing diagram of Figure 16.17(b). At time t_2, the listener deactivates the ack signal. It then senses the activation of the req signal and mistakenly treats the condition as a new round of handshaking and responds accordingly. Thus, the same incoming request pulse will be incorrectly regenerated again. On the other hand, if the listener is too slow, the talker may start to send a new request when the ack signal is still asserted, as shown in the timing diagram of Figure 16.17(c). At time t_3, the talker mistakenly thinks that the handshaking is completed and starts a new round shortly after. Since the listener still processes the first request, the new request will be lost. These examples show that all steps are needed in the original four-phase handshaking protocol.

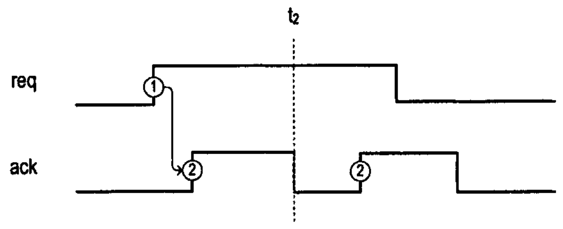
16.7.2 Two-phase handshaking protocol

In the four-phase handshaking protocol, the talker and the listener exchange information on two separate occasions. One is during the first half of the handshaking, activation and acknowledgment of the req signal, and the other is during the second half, deactivation and acknowledgment of the req signal. Some applications, such as sending an enable pulse across the domain, require only a single exchange of information. In these applications, the req signal (e.g., the enable signal) has already been successfully detected and regenerated in the first half. The purpose of the second half is to ensure that the system can return safely to the initial state.

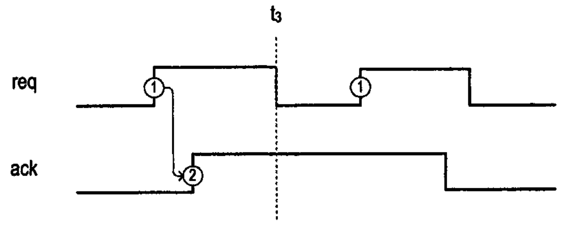
We can make the handshaking scheme more efficient by including only a single information exchange in the protocol. In this scheme, we do not require the system to return to the original state and define that the system is idle when the req and ack signals are



(a) Ideal scenario



(b) Error due to slow talker response



(c) Error due to slow listener response

Figure 16.17 Timing diagrams of an erroneous protocol.

both '0' or both '1'. The system will alternate between the two representations of the idle state.

The operation sequence of the new protocol includes the following steps:

1. The talker activates the `req` signal.
2. When the listener detects activation of the `req` signal, it activates the `ack` signal to inform the talker.
3. After the talker senses activation of the `ack` signal, it knows that the handshaking is completed and the system reaches the idle state.

Note that the values of the `req` and `ack` signals are "11". When a new round of handshaking is initiated, the system starts from the "11" state and the steps are:

1. The talker deactivates the `req` signal.
2. When the listener detects deactivation of the `req` signal, it deactivates the `ack` signal to inform the talker.
3. After the the talker senses deactivation of the `ack` signal, it knows that the handshaking is completed and the system reaches the idle state.

Note that the values of the `req` and `ack` signals are "00" now, and thus the system returns to its initial state. The timing diagram is shown in Figure 16.18. Although the appearance of the four-phase and two-phase timing diagrams are similar, interpretation of the `req` and `ack` signals (i.e., system state) is very different.

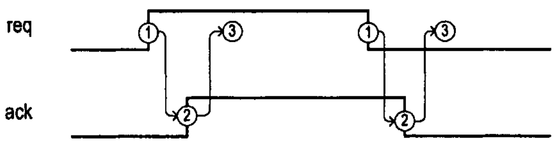


Figure 16.18 Timing diagram of the two-phase handshaking protocol.

We can follow the previous procedure to derive the talker and listener FSMs for the two-phase handshaking protocol. The revised the talker and listener ASM charts are shown in Figure 16.19. Note that the talker FSM stays in the `s_req1` and `s_req0` states until a new round of handshaking is initiated (i.e., when the `start` signal is '1'). Closer observation shows that the `idle` and `s_req0` states of the talker FSM are equivalent, and we can merge the two states and remove the `idle` state.

As in the four-phase handshaking system, two synchronizers are needed for the acknowledge and request signals in the final implementation.

16.8 DATA TRANSFER CROSSING CLOCK DOMAINS

Data transfer between synchronous subsystems is just passing data from one register to another register, and the operation takes one clock cycle. Data transfer between two clock domains is more complicated. As in passing a single enable signal, it involves two issues, which are synchronization of the data signals and regulation of the data transfer rate.

In most applications, the interface between clock domains includes command signals, data lines and address lines. As we discussed in Section 16.5.5, synchronizing related signals is difficult and error-prone. A better alternative is to bundle all signals and use an enable signal to coordinate the access of the bundled signals. Basically, the sending subsystem activates the bundled signals, waits until they are stabilized, and then activates an enable signal to inform the receiving subsystem to access the bundled signals. Since the bundled signals are stabilized when accessed, no timing violation will occur. Only the enable signal is subjected to the metastability condition and needs to be synchronized. Instead of worrying about the synchronization of all signals, we need only focus on the enable signal.

Since clock frequencies and data processing rates are likely to be different in two clock domains, resolving the synchronization problem alone cannot guarantee reliable data transfer. We also need a mechanism to control the rate of data transfer to ensure that no information is lost or duplicated during the transaction. We can incorporate the data transfer into the earlier handshaking protocols and divide the transfer into three categories:

- Four-phase handshaking transfer
- Two-phase handshaking transfer
- One-phase transfer

The four-phase handshaking transfer has the highest overhead but is most robust. It assumes that the two subsystems have minimal information about each other. One-phase transfer uses a single enable signal with no feedback. It involves minimal overhead, but its operation is based on the assumption that the two subsystems have prior knowledge of the other's timing characteristics.

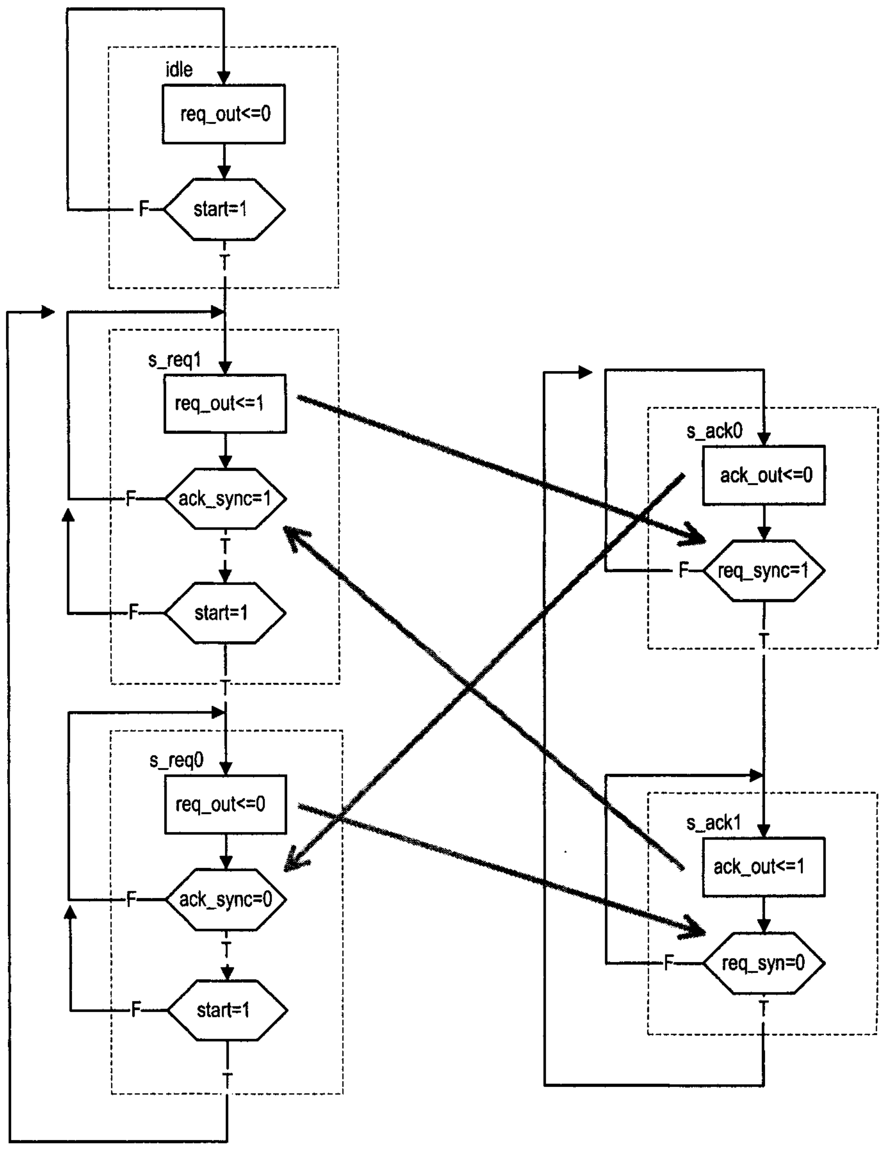
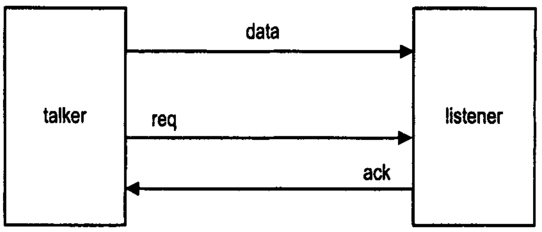


Figure 16.19 ASM charts of the talker and listener of the two-phase handshaking protocol.



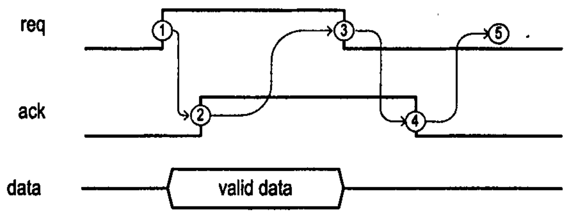


Figure 16.20 Push operation using the four-phase handshaking protocol.

For an asynchronous subsystem, storing data into another subsystem is known as a *push* operation and retrieving data from another subsystem is known as a *pull* operation. Many applications process data stage after stage, and thus the push operation is more common.

16.8.1 Four-phase handshaking protocol data transfer

The `req` and `ack` signals of the handshaking protocol form a special signaling mechanism and can be associated with various operations in the talker and listener. They can be used to perform push, pull or combined operations.

Basic one-direction data transfer Let us first consider the basic push operation, in which the talker transfers one data word to the listener. The conceptual block diagram and a representative timing diagram are shown in Figure 16.20. The basic handshaking sequence remains the same, and the talker places data on the data bus according to activation and deactivation of the `req` signal. The operation follows the basic handshaking sequence:

1. The talker activates the `req` signal and also places the data on the data bus.
2. The listener detects activation of the `req` signal and understands that data is available. After retrieving and processing the data, it activates the `ack` signal.
3. When the talker senses activation of the `ack` signal, it deactivates the `req` signal and removes the data from the data bus.
4. The listener deactivates the `ack` signal accordingly.
5. Once the talker senses deactivation of the `ack` signal, it knows the data transfer is completed and a new one can be initiated.

A possible implementation of the talker and listener is shown in Figure 16.21. We assume that the data line is a tri-state bus. The talker can place the data word on the bus by asserting the `tri_oe` signal, the enable signal of the tri-state buffer. As discussed above, the data is placed on the bus when the `req` signal is asserted. This can be achieved by asserting the `tri_oe` signal in the `s_req1` state of the talker FSM. Note that when the data is on the bus,

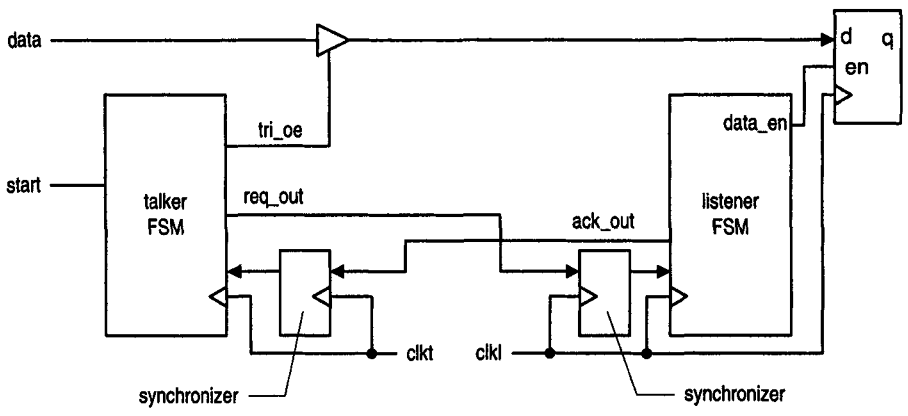


Figure 16.21 Block diagram of the push operation.

the `req_out` signal is also asserted. We can actually use the `req_out` signal to control the tri-state buffer.

The listener has a register for the input data and retrieves the data word by asserting the `data_en` signal, the enable signal of the register. The `data_en` signal can be asserted when the listener detects activation of the `req_syn` signal. Since the `req_syn` signal is delayed by two D FFs of the synchronizer, its activation is at least one clock cycle later than activation of the `req` and data signals. Thus, the data signal should be stabilized when the `req_sync` signal is activated and thus no timing violation will occur. We can modify the code of the listener FSM in Listing 16.8 to include the `data_en` signal as an output signal:

```
. . .
state_next <= state_reg;
data_en <='0';
case state_reg is
   when s_ack0 =>
      if req_sync='1' then
         state_next <= s_ack1;
         data_en <='1'; -- activate enable signal
      end if;
   when s_ack1 =>
      . . .
end case;
```

Note that this design is only for demonstration purposes. Since the data transfer is not bidirectional, the tri-state buffer is not actually needed. The push operation should function properly as long as the desired data is placed on the data bus when the `req_out` signal is asserted.

The basic pull operation is similar to the push operation except that the listener provides the data and the talker retrieves the data. The simplified block diagram and timing diagram are shown in Figure 16.22. After sensing activation of the `req` signal, the listener places the data on the data bus and activates the `ack` signal. Once detecting activation of the `ack` signal, the talker retrieves the data and deactivates the `req` signal. The listener then removes the data and deactivates the `ack` signal accordingly. Again, because the `ack_syn` signal is

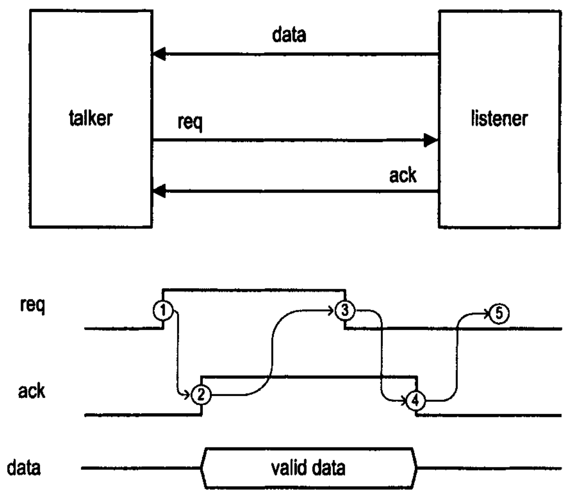


Figure 16.22 Pull operation using the four-phase handshaking protocol.

delayed by the synchronizer, the data signal should be stabilized when the `ack_sync` signal is activated.

Bidirectional data transfer The four-phase handshaking protocol can also incorporate more sophisticated operation. The talker can bundle additional information, such as the commands and address lines, push them to the listener and later pull the result back. The listener retrieves the bundled signals, processes the data according to the command and activates the `ack` signal when the operation is done. The following example illustrates the use of handshaking to access an eight-word register file in a different clock domain. We assume that a system consists of a processor and an I/O controller, which reside in different clock domains. The processor can read data from or write data to the eight-word register file of the I/O controller through an asynchronous interface based on the four-phase handshaking protocol. The talker and listener are in the processor's clock domain and the I/O controller's clock domain respectively. The basic block diagram is shown in Figure 16.23. To reduce the clutter, only the main components and connections of the data paths are shown.

In this system, the processor first checks the `ready` signal to ensure that the talker is not busy and then initiates the access by activating the `start` signal of the talker accordingly. When asserting the `start` signal, the processor also uses the `rw` signal to indicate the type of operation ('1' for read and '0' for write), places the address of the register file on the `addr` line and, in the case of a write operation, places data on the data line. After detecting the `start` signal, the talker of the asynchronous interface loads the address, the `rw` control signal and data (if needed) into its internal registers and starts the handshaking and data transfer operation.

The bundled signals include a 3-bit address line, a control signal, `pull`, and an 8-bit data line. Since the pull and push operations are mutually exclusive, the data line can be shared and thus is bidirectional.

The data path of the talker includes a register for the address, a register for the `rw` signal and two data registers to store the transmitted and received data. The data path of the listener is an eight-word register file. In a realistic scenario, the I/O controller should also be able

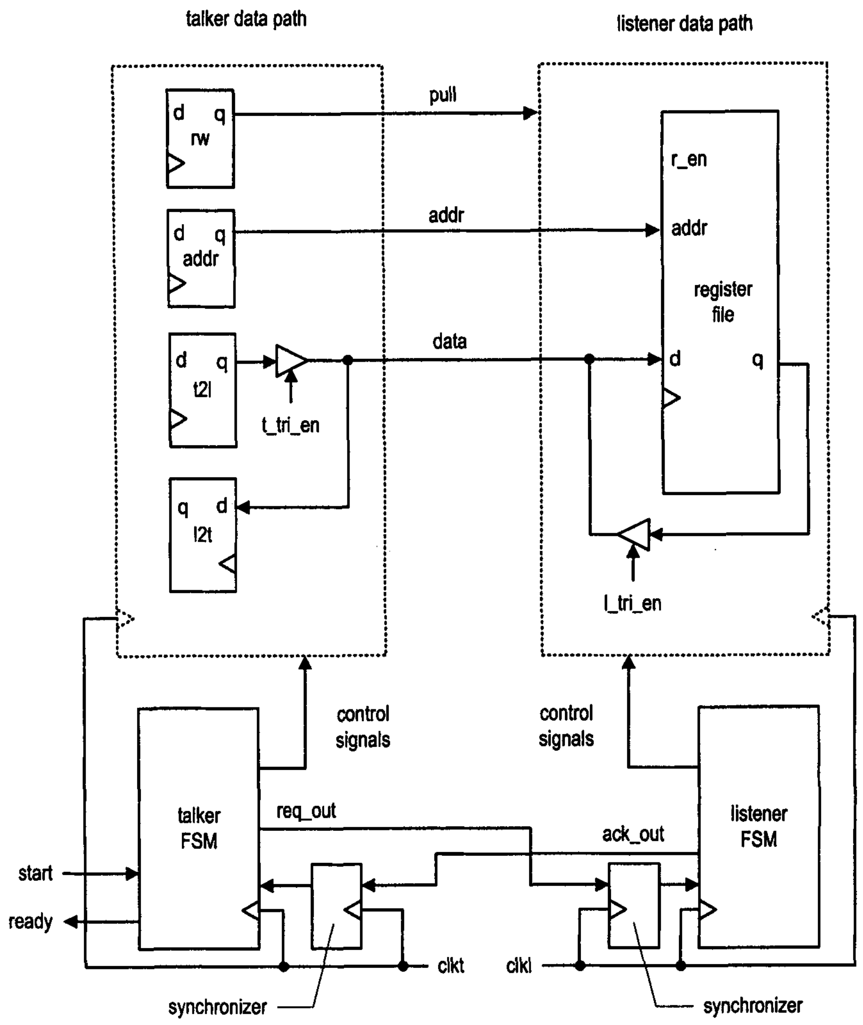


Figure 16.23 Block diagram of a push-and-pull system using the four-phase handshaking protocol.

to access the register file, and thus all signals should be multiplexed. For simplicity, the signals from the I/O controller to the register file are not shown.

The basic handshaking sequence of this circuit remains the same, and thus the state transition is similar to the FSMs of Section 16.7.1. The talker and listener FSMs also function as the control paths that control operation of the two data paths. In the `idle` state, the talker FSM checks the `start` signal. If it is asserted, the FSM moves to the `s_req1` state and stores the relevant information to the registers. The remaining operation of the data path depends on the type of access. Let us first consider the push operation. In the `s_req1` state, the talker activates the `req_out` signal and enables the tri-state buffer. The data is placed in the data line accordingly. Note that since the address line and the `pull` signal are not shared, they are connected to the listener data path during the entire operation.

When the listener FSM detects activation of the `req_sync` signal, it also checks the `pull` signal, whose '0' value indicates a push operation. At the next rising edge of the clock, the FSM moves to the `s_ack1` state, and the data will be stored into the location specified by the `addr` line. Note that the `req_sync` signal is delayed by the D FFs of the synchronizer. All other signals are already stabilized when its activation is detected. The FSM also activates the `ack_out` signal when entering the `s_ack1` state.

After the talker FSM senses activation of the `ack_sync` signal, it moves to the `s_req0` state, deactivates the `req_out` signal and disables the tri-state buffer. The talker and listener then proceed as in regular four-phase handshaking protocol to return to the initial state.

For the pull operation, the tri-state buffer of the talker is always disabled. When the listener FSM detects activation of the `req_sync` signal and assertion of the `pull` signal, it knows that the transaction is a pull operation. At the next rising edge of the clock, the listener FSM moves to the `s_ack1` state, activates the `ack_out` signal, and enables the tri-state buffer to place the register's output on the data line. When the talker FSM senses activation of the `ack_sync` signal, it knows that the data is also available. At the next rising edge of the clock, the talker FSM moves to the `s_req0` state, deactivates the `req_out` signal and stores the data into the `l2t` register. The talker and listener then proceed to return to the initial state.

The VHDL codes for the talker and listener interfaces are shown in Listings 16.11 and 16.12 respectively.

Listing 16.11 Talker interface of a push-and-pull system

```
library ieee;
use ieee.std_logic_1164.all;
entity talker_interface is
   port(
      clkt, resett: in std_logic;
      start, rw: in std_logic;  --read or write to i/o
      ack_sync: in std_logic;
      ready: out std_logic;
      req_out: out std_logic;
      d_t2l: in std_logic_vector(7 downto 0);
      addr_in: in std_logic_vector(1 downto 0);
      d_l2t: out std_logic_vector(7 downto 0);
      pull: out std_logic;
      addr: out std_logic_vector(1 downto 0);
      d: inout std_logic_vector(7 downto 0)
   );
end talker_interface;
```

```
architecture arch of talker_interface is
   type t_state_type is (idle, s_req1, s_req0);
   signal state_reg, state_next: t_state_type;
   signal req_buf_reg, req_buf_next: std_logic;
   signal t_tri_en: std_logic;
   signal l2t_next, l2t_reg: std_logic_vector(7 downto 0);
   signal t2l_next, t2l_reg: std_logic_vector(7 downto 0);
   signal rw_next, rw_reg: std_logic;
   signal addr_next, addr_reg: std_logic_vector(1 downto 0);
begin
   --================
   -- talker FSM
   --================
   -- state register and output buffer
   process(clkt,resett)
   begin
      if (resett='1') then
         state_reg <= idle;
         req_buf_reg <='0';
      elsif (clkt'event and clkt='1') then
         state_reg <= state_next;
         req_buf_reg <=req_buf_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,start,ack_sync)
   begin
      ready <='0';
      state_next <= state_reg;
      case state_reg is
         when idle =>
            if start='1' then
               state_next <= s_req1;
            end if;
            ready <= '1';
         when s_req1 =>
            if ack_sync='1' then
               state_next <= s_req0;
            end if;
         when s_req0 =>
            if ack_sync='0' then
               state_next <= idle;
            end if;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      case state_next is
         when idle =>
            req_buf_next <= '0';
         when s_req1 =>
         req_buf_next <= '1';
      when s_req0 =>
         req_buf_next <= '0';
   end case;
end process;
req_out <= req_buf_reg;
--==================
-- talker data path
--==================
-- data register
process(clkt,resett)
begin
   if (resett='1') then
      t2l_reg <= (others=>'0');
      l2t_reg <= (others=>'0');
      addr_reg <= (others=>'0');
      rw_reg <= '0';
   elsif (clkt'event and clkt='1') then
      t2l_reg <= t2l_next;
      l2t_reg <= l2t_next;
      addr_reg <= addr_next;
      rw_reg <= rw_next;
   end if;
end process;
-- data path next-state logic
process(state_reg,t2l_reg,l2t_reg,addr_reg,rw_reg,d_t2l,
        addr_in,d,rw,start,ack_sync)
begin
   t2l_next <= t2l_reg;
   l2t_next <= l2t_reg;
   addr_next <= addr_reg;
   rw_next <= rw_reg;
   t_tri_en <= '0';
   case state_reg is
      when idle =>
         rw_next <= rw;
         addr_next <= addr_in;
         if (start='1' and rw='0') then -- write to i/o
            t2l_next <= d_t2l;
         end if;
      when s_req1 =>
         if (rw_reg='0') then -- write to i/o
            t_tri_en <= '1';
         end if;
         if (ack_sync='1') and (rw_reg='1') then
          l2t_next <= d;
         end if;
      when s_req0 =>
         t_tri_en <= '0';
   end case;
end process;
-- output
d <= t2l_reg when t_tri_en='1' else (others=>'Z');
```

```
   pull <= rw_reg;
   d_l2t <= l2t_reg;
   addr <= addr_reg;
end arch;
```

Listing 16.12 Listener interface of a push-and-pull system

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity listener_interface is
   port(
      clkl, resetl: in std_logic;
      req_sync: in std_logic;
      ack_out: out std_logic;
      pull: in std_logic;
      addr: in std_logic_vector(1 downto 0);
      d: inout std_logic_vector(7 downto 0)
   );
end listener_interface;

architecture arch of listener_interface is
   type l_state_type is (s_ack0, s_ack1);
   signal state_reg, state_next: l_state_type;
   signal ack_buf_reg, ack_buf_next: std_logic;
   signal l_tri_en, r_en: std_logic;
   type r_file_type is array (3 downto 0) of
        std_logic_vector(7 downto 0);
   signal r_file_reg: r_file_type;
begin
   --===============
   -- listener FSM
   --===============
   -- state register and output buffer
   process(clkl,resetl)
   begin
      if (resetl='1') then
         state_reg <= s_ack0;
         ack_buf_reg <='0';
      elsif (clkl'event and clkl='1') then
         state_reg <= state_next;
         ack_buf_reg <= ack_buf_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,req_sync)
   begin
      state_next <= state_reg;
      case state_reg is
         when s_ack0 =>
            if req_sync='1' then
               state_next <= s_ack1;
            end if;
         when s_ack1 =>
            if req_sync='0' then
               state_next <= s_ack0;
            end if;
         end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      case state_next is
         when s_ack0 =>
            ack_buf_next <= '0';
         when s_ack1 =>
            ack_buf_next <= '1';
         end case;
   end process;
   ack_out<= ack_buf_reg;
   --===================
   -- listener data path
   --===================
   -- register file
   process(clkl,resetl)
   begin
      if (resetl='1') then
         for i in 0 to 3 loop
            r_file_reg(i) <= (others=>'0');
         end loop;
      elsif (clkl'event and clkl='1') then
         if r_en='1' then
            r_file_reg(to_integer(unsigned(addr))) <= d;
         end if;
     end if;
   end process;
   -- enable logic
   process(state_reg,req_sync,pull)
   begin
      l_tri_en <= '0';
      r_en <= '0';
      case state_reg is
         when s_ack0 =>
            if (req_sync='1')  then
               if (pull='0') then -- push
                  r_en <= '1';
               end if;
            end if;
         when s_ack1 =>
            if (pull='1')then
               l_tri_en <='1';
            end if;
      end case;
   end process;
    -- output
   d <= r_file_reg(to_integer(unsigned(addr)))
```

```
                           when l_tri_en='1' else
          (others=>'Z');
end arch;
```

As in the handshaking code of Section 16.7.1, we must add two synchronizers for the request and acknowledge signals to complete the implementation.

Performance of four-phase handshaking data transfer The strength of four-phase handshaking is that it makes a minimal assumption about the two subsystems. It will function properly even if a subsystem has no knowledge of the clock frequency and the data processing rate of other subsystems. However, there is a high overhead associated with this protocol. Assume that the clock period of the talker and listener are T_{c_t} and T_{c_l} respectively. We can estimate the required time to complete one data transfer. During a data transfer, each FSM traverses all its states and then returns to the initial state. Since the talker and listener FSMs have three and two states respectively, it takes $3T_{c\_t} + 2T_{c\_l}$. Because both the ack and req signals cross the clock domain, synchronizers are needed. If we assume that two-FF synchronizers are used, the synchronization requires up to two clock cycles whenever a signal is synchronized. The ack signal is used twice in the talker FSM, and the synchronization introduces an overhead of $4T_{c\_t}$. Similarly, synchronization of the req signal introduces an overhead of $4T_{c\_l}$. Thus, it takes $7T_{c\_t} + 6T_{c\_l}$ to complete one data transfer, which is very slow compared with the one-clock synchronous data transfer.

16.8.2 Two-phase handshaking data transfer

The two-phase handshaking protocol can reduce the overhead by half. However, since only a single handshaking occurs in the protocol, this scheme is less flexible and imposes certain constraints on the data transfer.

Let us first consider the push operation. The data transfer can be embedded in the two-phase handshaking protocol as follows:

1. The talker activates the req signal and places data on the data bus.
2. The listener detects activation of the req signal. It retrieves the data and activates the ack signal.
3. Once the talker senses activation of the ack signal, it removes the data from the data bus.

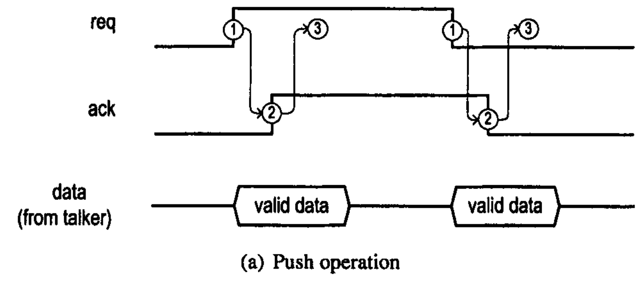
The first push operation is done at this point. Note that both the req and ack signals are '1'. When the talker wants to push the next data, the handshaking continues from this state:

1. The talker deactivates the req signal and places data on the data bus.
2. The listener detects deactivation of the req signal. It retrieves the data and deactivates the ack signal.
3. Once the talker senses deactivation of the ack signal, it removes the data from the data bus.

Note that after two push operations, the req and ack signals will be '0' and the system returns to the original state.

The block diagram for the two-phase push operation is identical to the four-phase push operation, as shown in Figure 16.20(a). A representative timing diagram is shown in Figure 16.24(a). Unlike the four-phase push operation, the req signal remains unchanged when the talker removes the data from the data bus.

Using the two-phase handshaking protocol to perform a pull operation is more difficult. The two-phase operation only allows the listener to signal the talker that it has placed the



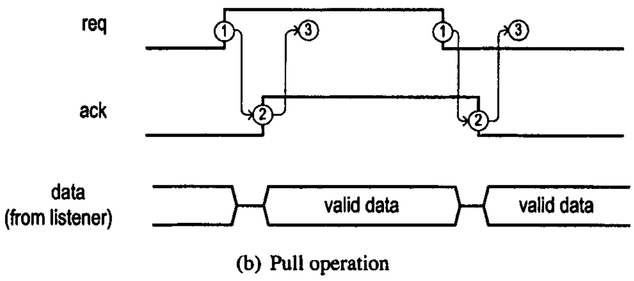


Figure 16.24 Timing diagrams of push and pull operations using two-phase handshaking protocol.

data on the data bus. There is no explicit signaling mechanism to let the listener know when the data is retrieved and when the data can be removed from the bus. One way to overcome the problem is to embed this information in the next operation. When the talker initiates a new data transfer, it implicitly indicates that the data from the previous pull operation has been retrieved. Thus, when the listener detects the transition of the `req` signal of the next operation, it can safely remove the data from the data bus. The timing diagram is shown in Figure 16.24(b). Note that data must stay on the data bus for a long time if the two pull operations are far apart.

In the four-phase handshaking protocol, the two handshaking operations are used to indicate the initiation and completion of the data transfer. The values of the `req` and `ack` signals represent the system state and can be used to indicate the status of the data line as well. The talker and the listener, or any other subsystems that have access to the two signals, can determine the status of the data line via the two signals. This feature is important if the data line is shared. For example, in the push-and-pull design of Section 16.8.1, the push and pull are done in the same data line. The talker and listener need to know the status of the line to avoid bus fighting. On the other hand, in the two-phase handshaking protocol, handshaking operation is used only for initiation of the data transfer. The status of the data line cannot be determined by the `req` and `ack` signals, and thus the line cannot be shared. If we want to use the two-phase handshaking protocol for the previous push–pull design, separate data lines are needed for the push and pull operations.

16.8.3 One-phase data transfer

If the characteristics of the listener are known in advance, we can customize the data transfer timing and eliminate the acknowledge signal. Since there is no feedback, the request signal behaves like the enable pulse discussed in Section 16.6.

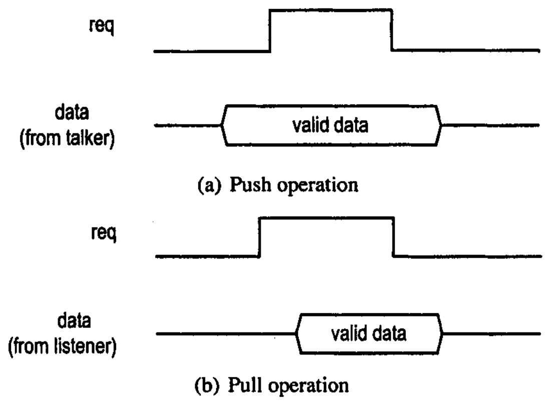


Figure 16.25 Timing diagrams of push and pull operations using one-phase protocol.

Let us first consider the push operation. The `req` signal now functions as an enable signal to inform the listener of the availability of the data. Since there is no feedback from the listener, the talker relies on prior knowledge about the listener to calculate the minimal assertion time for the data signal. The talker asserts the `req` signal and places the data on the data bus for a predetermined interval. The listener will detect activation of the `req` signal and retrieve the data within this interval. We can use the schemes discussed in Section 16.6 to regenerate an enable pulse from the `req` signal. A representative timing diagram is shown in Figure 16.25(a). If we assume that the listener is always available, the listener needs about three clock cycles (i.e., $3T_{c\_l}$) to store the data into a register. The interval includes two clock cycles to synchronize the `req` signal and one clock cycle to store the data.

The basic pull operation can be done in a similar fashion. The listener knows in advance how long the data should be put on the data line, and the talker knows when the data should be available. After activating the `req` signal, the talker waits for a predetermined amount of time and then retrieves data from the data line. A representative timing diagram is shown in Figure 16.25(b).

16.9 DATA TRANSFER VIA A MEMORY BUFFER

Although the handshaking protocol provides a reliable mechanism to transfer data across clock domains, it is not an efficient scheme. Each transaction involves a large overhead, and thus this method is good only for small, random exchanges of information between two subsystems. It is not an effective way to move a large amount of data between the two clock domains. A better alternative is to use a memory buffer as temporary storage. Instead of direct interactions, the two subsystems store and retrieve data via the memory buffer. Two common configurations are the *asynchronous FIFO buffer* and *shared memory*. These configurations cannot eliminate the metastable condition but can significantly reduce the overhead associated with data transfer.

16.9.1 FIFO buffer

A FIFO buffer is like a one-directional pipe. The sending subsystem puts the data in one end of the pipe, and the receiving subsystem retrieves the data from the other end of the pipe. In Section 9.3.2, we discussed the operation and design of a synchronous FIFO buffer, in which the two subsystems are controlled by the same clock signal. The operation of an asynchronous FIFO is similar, but the sending and receiving subsystems are controlled by clocks from different clock domains.

In an asynchronous FIFO, the read pointer (counter) is controlled by the clock signal from the receiving subsystem and the write pointer (counter) is controlled by the clock signal from the sending subsystem. Since the operation of these counters only involves the clock signal from its own domain, the counters impose no synchronization problem. The difficulty comes from the `full` and `empty` status signals. As discussed in Section 9.3.2, there are several different methods to obtain the status. These methods need information from both the sending and receiving subsystems and thus involve the signals from two clock domains. The main task of implementing an asynchronous FIFO is to design a circuit that generates reliable, properly synchronized status signals.

One possible implementation is to follow the synchronous FIFO organization discussed in Figure 9.14. For a synchronous FIFO with an n-bit address space (i.e., 2^n words), it is constructed as follows:

- Use two $(n + 1)$-bit binary counters as the pointers, one for the read pointer and one for the write pointer.
- Use two n-bit binary counters (which are the n LSBs of the $(n + 1)$-bit counters) as the read and write addresses to access the designated element of the memory array.
- Compare the two $(n + 1)$-bit counters to obtain full and empty status.

To use this scheme in an asynchronous environment, we must ensure that the comparison circuit can generate the full and empty status signals that are synchronized with their respective clock domains. To accomplish this, we must revise the design as follows:

- Add a synchronizer in the comparison circuit to synchronize the pointer from the other clock domain.
- Replace $(n + 1)$-bit and n-bit binary counters with the $(n + 1)$-bit and n-bit Gray counters.

In synchronous FIFO, the read and write pointers are implemented by binary counters or LFSRs. In these counters, there may be multiple bit changes in a transition. For example, consider a 4-bit binary counter. When the counter wraps around from "1111" to "0000", all four bits change. As discussed in Section 16.5.5, synchronizing multiple changing bits may lead to the capture of erroneous, intermediate transition values, and thus these counters can cause problems if the values are passed to a different clock domain. To prevent this, we must use a Gray counter for the pointer, in which only one bit is changed in a transition. The circulation pattern of a 4-bit counter is shown in the first column of Table 16.2.

In Section 9.3.2, we add an extra bit in the binary counter and use this bit (the MSB of the counter) to distinguish whether the FIFO is full or empty. In this approach, we use two $(n+1)$-bit binary counters as the read and write pointers, and use two n-bit binary counters for the write and read addresses of the memory array. Note that the MSB of the Gray counter is the same as the MSB of the binary counter, and thus it can be used to distinguish whether the FIFO is empty or full. As in the binary counter–based implementation, we need two $(n + 1)$-bit Gray counters as the pointers and two n-bit Gray counters as the addresses.

It is straightforward to obtain the n-bit binary counting patterns since they are the same as the n LSBs of the $(n+1)$-bit binary counter. It is more difficult for the Gray counter. For example, the counting patterns of three LSBs of the 4-bit Gray counter and the counting patterns of a 3-bit Gray counter are shown in the second and third columns of Table 16.2. Their patterns are different in the bottom half. Although the patterns are not identical, there

Table 16.2 Circulation pattern of 4-bit and 3-bit Gray counters

4-bit Gray counter	3 LSBs of 4-bit Gray counter	3-bit Gray counter
0000	000	000
0001	001	001
0011	011	011
0010	010	010
0110	110	110
0111	111	111
0101	101	101
0100	100	100
1100	100	000
1101	101	001
1111	111	011
1110	110	010
1010	010	110
1011	011	111
1001	001	101
1000	000	100

is no need to construct a separate n-bit Gray counter from scratch. Closer observation shows that the $(n-1)$ LSBs of the $(n+1)$-bit Gray counter and n-bit Gray counter are identical, and the MSB of the n-bit Gray counter can be obtained by performing an xor operation on the two MSBs of the $(n+1)$-bit Gray counter. In other words, let $a_n, a_{n-1}, \ldots, a_0$ be the bits of an $(n+1)$-bit Gray counter, and $b_{n-1}, b_{n-2}, \ldots, b_0$ be the bits of an n-bit Gray counter. We can derive the n-bit counting pattern by using

$$b_i = \begin{cases} a_{i+1} \oplus a_i & \text{if } i = n-1 \\ a_i & \text{otherwise} \end{cases}$$

The block diagram of an n-bit asynchronous FIFO control circuit is shown in Figure 16.26. In the write control part, an $(n+1)$-bit Gray counter is used as the write pointer and the derived n-bit Gray counter is used for the write address. The read pointer is obtained from the read control part. It is first synchronized by an $(n+1)$-bit synchronizer. The comparing circuit derives the n-bit read address and compares it to the write address. If the read and write addresses are the same and the MSBs of the read and write pointers are different, the FIFO is full and the `full` signal is asserted accordingly. Since all inputs of the comparing circuits are synchronized with the write controller's clock, the `full` signal will not cause a timing violation when used. The read control part essentially mirrors the write control except for the minor difference in the comparing circuit. The `empty` signal will be asserted when the read and write addresses are the same and the MSBs of the read and write pointers are the same.

The VHDL codes of the write port control and the read port control are shown in Listings 16.13 and 16.14 respectively. The code of the Gray counter is similar to the code discussed in Section 7.5.1. We use a generic, `N`, to express the number of bits of the FIFO control circuit. The code of a generic n-bit two-FF synchronizer is shown in Listing 16.15.

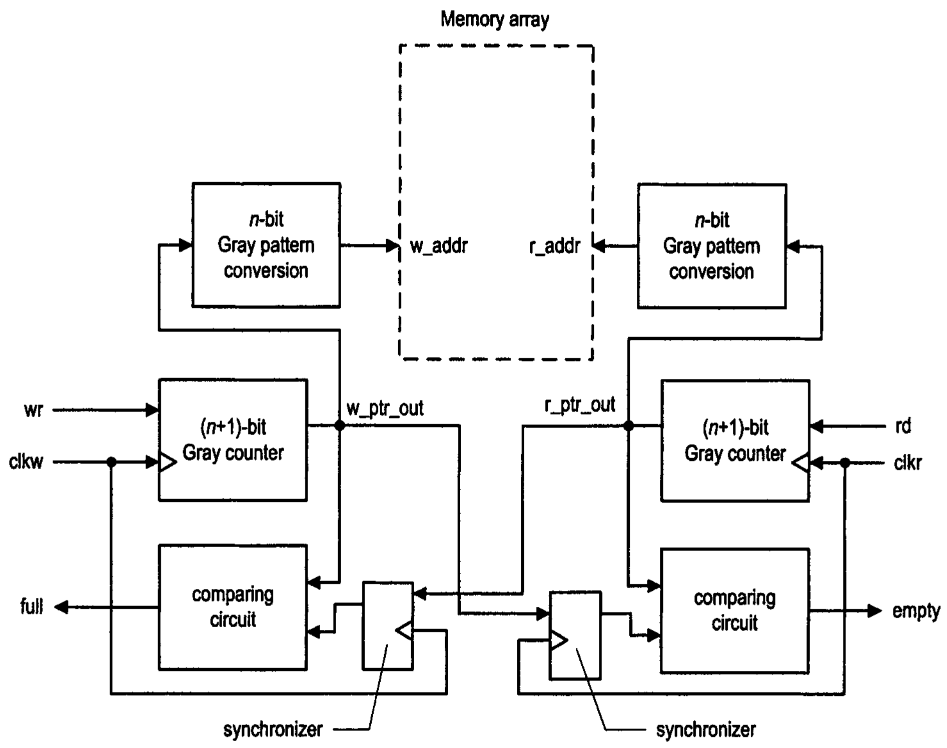


Figure 16.26 Block diagram of an asynchronous FIFO controller.

The complete asynchronous FIFO control circuit follows the basic block diagram, and its VHDL code is shown in Listing 16.16.

Listing 16.13 Write port control of an asynchronous FIFO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_write_ctrl is
   generic(N: natural);
   port(
      clkw, resetw: in std_logic;
      wr: in std_logic;
      r_ptr_in: in std_logic_vector(N downto 0);
      full: out std_logic;
      w_ptr_out: out std_logic_vector(N downto 0);
      w_addr: out std_logic_vector(N-1 downto 0)
   );
end fifo_write_ctrl;

architecture gray_arch of fifo_write_ctrl is
   signal w_ptr_reg, w_ptr_next:
      std_logic_vector(N downto 0);
   signal gray1, bin, bin1: std_logic_vector(N downto 0);
   signal waddr_all: std_logic_vector(N-1 downto 0);
   signal waddr_msb, raddr_msb: std_logic;
   signal full_flag: std_logic;
begin
   -- register
   process(clkw,resetw)
   begin
      if (resetw='1') then
          w_ptr_reg <= (others=>'0');
      elsif (clkw'event and clkw='1') then
         w_ptr_reg <= w_ptr_next;
      end if;
   end process;
   -- (N+1)-bit Gray counter
   bin <= w_ptr_reg xor ('0' & bin(N downto 1));
   bin1 <= std_logic_vector(unsigned(bin) + 1);
   gray1 <= bin1 xor ('0' & bin1(N downto 1));
   -- update write pointer
   w_ptr_next <= gray1 when wr='1' and full_flag='0' else
                 w_ptr_reg;
   -- N-bit Gray counter
   waddr_msb <=  w_ptr_reg(N) xor w_ptr_reg(N-1);
   waddr_all <= waddr_msb & w_ptr_reg(N-2 downto 0);
   -- check for FIFO full
   raddr_msb <= r_ptr_in(N) xor r_ptr_in(N-1);
   full_flag <=
     '1' when r_ptr_in(N) /=w_ptr_reg(N) and
         r_ptr_in(N-2 downto 0)=w_ptr_reg(N-2 downto 0) and
         raddr_msb = waddr_msb else
     '0';
   -- output
   w_addr <= waddr_all;
   w_ptr_out <= w_ptr_reg;
   full <= full_flag;
end gray_arch;
```

Listing 16.14 Read port control of an asynchronous FIFO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_read_ctrl is
   generic(N: natural);
   port(
      clkr, resetr: in std_logic;
      w_ptr_in: in std_logic_vector(N downto 0);
      rd: in std_logic;
      empty: out std_logic;
      r_ptr_out: out std_logic_vector(N downto 0);
      r_addr: out std_logic_vector(N-1 downto 0)
   );
end fifo_read_ctrl;

```

```
architecture gray_arch of fifo_read_ctrl is
   signal r_ptr_reg, r_ptr_next: std_logic_vector(N downto 0);
   signal gray1, bin, bin1: std_logic_vector(N downto 0);
   signal raddr_all: std_logic_vector(N-1 downto 0);
   signal raddr_msb,waddr_msb: std_logic;
   signal empty_flag: std_logic;
begin
   -- register
   process(clkr,resetr)
   begin
      if (resetr='1') then
         r_ptr_reg <= (others=>'0');
      elsif (clkr'event and clkr='1') then
         r_ptr_reg <= r_ptr_next;
      end if;
   end process;
   -- (N+1)-bit Gray counter
   bin <= r_ptr_reg xor ('0' & bin(N downto 1));
   bin1 <= std_logic_vector(unsigned(bin) + 1);
   gray1 <= bin1 xor ('0' & bin1(N downto 1));
   -- update read pointer
   r_ptr_next <= gray1 when rd='1' and empty_flag='0' else
                 r_ptr_reg;
   -- N-bit Gray counter
   raddr_msb <= r_ptr_reg(N) xor r_ptr_reg(N-1);
   raddr_all <= raddr_msb & r_ptr_reg(N-2 downto 0);
   waddr_msb <= w_ptr_in(N) xor w_ptr_in(N-1);
   -- check for FIFO empty
   empty_flag <=
      '1' when w_ptr_in(N)=r_ptr_reg(N) and
          w_ptr_in(N-2 downto 0)=r_ptr_reg(N-2 downto 0) and
          raddr_msb = waddr_msb else
      '0';
    -- output
    r_addr <= raddr_all;
    r_ptr_out <= r_ptr_reg;
    empty <= empty_flag;
end gray_arch;
```

Listing 16.15 *n*-bit synchronizer

```
library ieee;
use ieee.std_logic_1164.all;
entity synchronizer_g is
   generic(N: natural);
   port(
      clk, reset: in std_logic;
      in_async: in std_logic_vector(N-1 downto 0);
      out_sync: out std_logic_vector(N-1 downto 0)
   );
end synchronizer_g;

architecture two_ff_arch of synchronizer_g is
```

```
   signal meta_reg, sync_reg: std_logic_vector(N-1 downto 0);
   signal meta_next, sync_next:
      std_logic_vector(N-1 downto 0);
begin
   -- two registers
   process(clk,reset)
   begin
      if (reset='1') then
         meta_reg <= (others=>'0');
         sync_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         meta_reg <= meta_next;
         sync_reg <= sync_next;
      end if;
   end process;
   -- next-state logic
   meta_next <= in_async;
   sync_next <= meta_reg;
   -- output
   out_sync <= sync_reg;
end two_ff_arch;
```

Listing 16.16 Top-level structural description of an asynchronous FIFO control circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_async_ctrl is
   generic(DEPTH: natural);
   port(
      clkw: in std_logic;
      resetw: in std_logic;
      wr: in std_logic;
      full: out std_logic;
      w_addr: out std_logic_vector (DEPTH-1 downto 0);
      clkr: in std_logic;
      resetr: in std_logic;
      rd: in std_logic;
      empty: out std_logic;
      r_addr: out std_logic_vector (DEPTH-1 downto 0)
   );
end fifo_async_ctrl ;

architecture str_arch of fifo_async_ctrl is
   signal r_ptr_in: std_logic_vector(DEPTH downto 0);
   signal r_ptr_out: std_logic_vector(DEPTH downto 0);
   signal w_ptr_in: std_logic_vector(DEPTH downto 0);
   signal w_ptr_out: std_logic_vector(DEPTH downto 0);
   -- component declarations
   component fifo_read_ctrl
      generic(N: natural);
      port(
         clkr: in std_logic;
         rd: in std_logic;
         resetr: in std_logic;
         w_ptr_in: in std_logic_vector (N downto 0);
         empty: out std_logic;
         r_addr: out std_logic_vector (N-1 downto 0);
      r_ptr_out: out std_logic_vector (N downto 0)
      );
   end component;
   component fifo_write_ctrl
      generic(N: natural);
      port(
         clkw: in std_logic;
         r_ptr_in: in std_logic_vector (N downto 0);
         resetw: in std_logic;
         wr: in std_logic;
         full: out std_logic;
         w_addr: out std_logic_vector (N-1 downto 0);
         w_ptr_out: out std_logic_vector (N downto 0)
      );
   end component;
   component synchronizer_g
      generic(N: natural);
      port(
         clk: in std_logic;
         in_async: in std_logic_vector (N-1 downto 0);
         reset: in std_logic;
         out_sync: out std_logic_vector (N-1 downto 0)
      );
   end component;
begin
   read_ctrl: fifo_read_ctrl
      generic map(N=>DEPTH)
      port map (clkr=>clkr, resetr=>resetr, rd=>rd,
                w_ptr_in=>w_ptr_in, empty=>empty,
                r_ptr_out=>r_ptr_out, r_addr=>r_addr);
   write_ctrl: fifo_write_ctrl
      generic map(N =>DEPTH)
      port map(clkw=>clkw, resetw=>resetw, wr=>wr,
               r_ptr_in=>r_ptr_in, full=>full,
               w_ptr_out=>w_ptr_out, w_addr=>w_addr);
   sync_w_ptr: synchronizer_g
      generic map(N=>DEPTH+1)
      port map(clk=>clkw, reset=>resetw,
               in_async=>w_ptr_out, out_sync=>w_ptr_in);
   sync_r_ptr: synchronizer_g
      generic map(N=>DEPTH+1)
      port map(clk=>clkr, reset=>resetr,
               in_async=>r_ptr_out, out_sync =>r_ptr_in);
end str_arch;
```

Because of the synchronizer, the onset of the empty and full signals may be delayed. For example, assume that the FIFO is originally empty. After a write operation is performed, the write pointer changes and the FIFO contains one data item. It takes two clock cycles to propagate the change through the two cascading FFs of the synchronizer, and thus the onset of the `empty` signal is delayed by two clock cycles. During this interval, the reading subsystem will falsely assume that there is no data to retrieve and will stay idle. While the delay causes late data retrieval, the functionality of the FIFO remains intact and no invalid data item is retrieved though the buffer. The same situation happens to the `full` signal. The deassertion of the `full` signal is delayed by two clock cycles. During this interval, the sending subsystem falsely assumes that the FIFO is full and suspends the write operation.

The delayed `empty` and `full` signals force the subsystems to be idle unnecessarily and thus penalize the performance. The penalty is essentially due to the overhead associated with the synchronization of two clock domains and cannot be avoided. However, the idle situation occurs only when the FIFO is almost empty or almost full. There is no overhead or extra delay when the FIFO is partially full. In comparison, the handshaking scheme involves the overhead in every data transaction, and thus the FIFO buffer is more efficient.

16.9.2 Shared memory

Another frequently used buffering scheme is shared memory. The basic idea is to allow multiple subsystems to access a common memory. The sending subsystem can first write the data into the memory, and the receiving subsystem then obtains the data by reading the same memory location. This scheme is best suited when a large chunk of data, such as a high-resolution image, has to be transferred.

The shared memory scheme can be implemented by using a regular single-port memory or a special dual-port memory. For a single-port memory configuration, we can treat the memory as the shared resource and use an arbiter to resolve the conflicting requests and coordinate the memory usage. The basic design of the arbiter is similar to that in Section 10.8.2. Because the interactions are between different clock domains, synchronization circuits are needed for all request and grant signals.

The request-grant process is somewhat like the handshaking procedure and has a similar overhead. However, the overhead is associated with each resource arbitration, not each data transaction. Since one round of arbitration allows any amount of data to be transferred (up to the size of the shared memory), the average overhead of a single data transaction becomes very small.

A better alternative is to use dual-port memory. A dual-port memory has two independent access ports, each containing its own address line, data line and control signals. The multiplexing and decoding circuits are duplicated inside the memory module. Two memory accesses can be performed simultaneously as long as the memory addresses are different. A conflict occurs when two memory operations access the same memory location (i.e., two operations have the same memory addresses). An arbiter is used to resolve the condition. When there is no clock in the regular dual-port memory, the internal arbitrator is an asynchronous sequential circuit. It may enter a metastable state if the two request signals are asserted too closely. As in the synchronizer, the arbitration circuit needs to provide some time to resolve the metastable condition and thus introduces similar overhead.

As with other memory modules, dual-port memory cannot be synthesized from scratch in RT-level code. We must instantiate the predesigned module from the target device technology.

16.10 SYNTHESIS OF A MULTIPLE-CLOCK SYSTEM

The synchronous design is the most important methodology and the cornerstone of the entire design and fabrication process. The synthesis, timing analysis, verification and testing of synchronous systems are well understood, and many EDA software tools have been developed to automate the tasks.

One major motivation behind the synchronous methodology is to provide a systematic way to satisfy the timing constraints. The objective of synthesis and verification is to *identify* and *prevent* timing violations. The EDA software tools are developed to assist designers in achieving this objective. On the other hand, the metastability analysis and synchronization circuit are to deal with the scenario that a timing violation *has already occurred*. This is essentially a transistor-level phenomenon, and its behavior cannot easily be modeled at the gate or RT level. The EDA tools are not able to handle metastability, and the analysis and synthesis process cannot be automated. Most software can only detect and warn the onset of a time violation but cannot model or analyze what happens afterward. For example, in the `std_logic` data type, the `'X'` value is used to model the output after a timing violation. After a timing violation occurs, the output `'X'` value will be permanent and propagated to the downstream circuit. This is very different from the actual timing violation, in which the output signal may become '0' or '1', or be resolved to a stable value after a random period of time.

While the design considerations for a multiple-clock system are different from those of a synchronous system, it is not wise to abandon the synchronous design methodology and start from scratch. Instead, we want to incorporate the methodology into the new design flow and utilize the previous techniques and EDA tools as much as possible.

To achieve this goal, a multiple-clock system should be divided into synchronous subsystems and crossing-domain interfaces. A synchronous subsystem is within the same clock domain, and thus we can design it just as a regular synchronous system. On the other hand, the crossing domain interface involves the synchronization and data transfer protocol. Its analysis and design are very different from those of the synchronous system, and very few EDA tools are available for these tasks. We usually have to manually analyze, design and verify the interface circuit and protocols. Since the synchronization circuit depends on the device characteristics and the data transfer protocol sometimes depends on the clock rates of the domains, the interface is usually device dependent and is not portable.

The general design approach for a multiple-clock system can be summarized as follows:

1. Partition the system into locally synchronous subsystems.
2. Design and verify these subsystems following the synchronous methodology.
3. Develop protocol to pass data and exchange information between clock domains.
4. Manually analyze and design the necessary synchronization circuits between clock domains.
5. Verify operation of the entire system.

A representative top-level partition of a system with two clock domains is shown in Figure 16.27. It is a good idea to treat the synchronization circuit and data transfer interface as separate modules and instantiate them individually in VHDL code. These modules are normally device dependent and may need to be reanalyzed and redesigned when the system is ported to a different device technology or operation environment (e.g., a different clock rate).

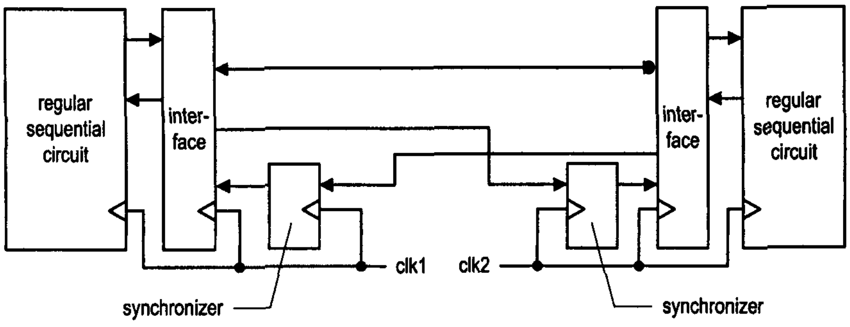


Figure 16.27 Partition of a system with two clock domains.

16.11 SYNTHESIS GUIDELINES

16.11.1 Guidelines for general use of a clock

- Do not manipulate the clock signal in regular RT-level design and synthesis.
- Minimize the number of clock signals in a system.
- Minimize the number of clock domains (i.e., the number of independent clock signals). Use a derived clock signal when possible.
- If a derived clock signal is needed, manually derive and instantiate the circuit and separate it from the regular synthesis.

16.11.2 Guidelines for a synchronizer

- Synchronize a signal in a single place.
- Avoid synchronizing related signals.
- Use a glitch-free signal for synchronization.
- Reanalyze and examine the synchronizer and MTBF when the device is changed or the clock rate is revised.

16.11.3 Guidelines for an interface between clock domains

- Clearly identify the boundary of the clock domain and the signals that cross the domain.
- Separate the synchronization circuits and asynchronous interface from the synchronous subsystems and instantiate them as individual modules.
- Use a reliable synchronizer design to provide sufficient metastability resolution time.
- Analyze the data transfer protocol over a wide range of scenarios, including faster and slower clock frequencies and different data rates.

16.12 BIBLIOGRAPHIC NOTES

The construction of the clock network involves the transmission and distribution of electronic signals. It is normally covered under the subjects of signal integration and high-speed design. Two texts by Howard Johnson, *High-Speed Digital Design: A Handbook of Black Magic* and *High-Speed Signal Propagation: Advanced Black Magic*, provide comprehensive coverage of this topic.

The study of metastability and synchronization failure relies on transistor-level model and analysis. The text, *Digital Systems Engineering* by William J. Dally and John W. Poulton, covers the theoretical foundation of this subject. The book also includes a discussion of the design of asynchronous circuit.

The more practical design materials on the synchronizer and asynchronous interface can be found in manufacturers' application notes or articles from technical conferences. Articles by Clifford E. Cummings, *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*, *Simulation and Synthesis Techniques for Asynchronous FIFO Design* and *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*, provide many practical design examples and good advice.

Problems

16.1 Assume that a sequential system with an ideal clock signal can operate at a maximal clock rate of 100 MHz. If the physical clock distribution network introduces a 1.5-ns clock skew, what is the new maximal clock rate?

16.2 Consider a D FF with w and τ.

- **(a)** If we improve the D FF by reducing w by 10%, discuss the effect on MTBF.
- **(b)** If we improve the D FF by reducing τ by 10%, discuss the effect on MTBF.

16.3 At the end of Section 16.4.2, we discuss the difference between T_r and T_{r2}. Assume that w and τ are identical for the calculation of MTBF(T_r) and MTBF(T_{r2}). Derive MTBF(T_r) in terms of MTBF(T_{r2}), w and τ.

16.4 For the two-FF synchronizer discussed in Section 16.5.3, determine the new MTBF for the following:

- **(a)** The placement and routing process adds a 2.5-ns wiring delay.
- **(b)** The system clock rate is decreased by 10%.
- **(c)** The setup time of the D FF is reduced by 10%.
- **(d)** The setup time of the D FF is reduced by 25%.
- **(e)** The τ of the D FF is reduced by 10%.
- **(f)** The τ of the D FF is reduced by 25%.

16.5 We want to regenerate the enable pulse in the listener's clock domain using the four-phase handshaking protocol. In this scheme, the listener has an output signal that is asserted once during the handshaking process.

- **(a)** Revise the listener ASM chart of Figure 16.16 to add a Mealy output signal.
- **(b)** Modify the VHDL code to reflect the revised ASM chart.

16.6 Repeat Problem 16.5, but add a Moore output signal.

16.7 Repeat Problem 16.5 for the two-phase handshaking protocol of Figure 16.19.