# Internet Addendum
# Sampling and detecting pulses

## An introductory note: the power-on reset

Be aware that, for the sake of clarity, in the circuits shown here, the power-on reset has been omitted. It is obvious to begin by clearing all flip-flops. In finite state machines (FSMs) that encode each state by its own flip-flop (one-hot encoding, OHE), the idle state must be preset and all others cleared. Usually, this is done via the flip-flop's preset and clear inputs. This is, however, conventional textbook wisdom that does not always apply when designing for CPLDs and FPGAs. It is not always expedient to initialize all flip-flops. There are subtleties regarding effective use of the circuit's resources, synchronization, and so on. I recommend consulting the appropriate manuals and application notes (like, for example, [22], [A1], and [A2]). The references [A1] to [A4] are to be found at the end of this addendum.

## Synchronizing narrow pulses

Narrow pulses are to be synchronized with a sampling clock. To do this, pulse memory and synchronizer flip-flops are combined. The incoming pulse is instantly captured and held in the pulse memory flip-flop until the next sampling cycle. As soon as the data has been clocked into the synchronization flip-flop, the pulse memory will be cleared. Figure 1 shows two circuits with different precedence behavior. If the input pulses occur at sufficiently wide intervals, both circuits will emit an output pulse one clock cycle wide (Figure 2).

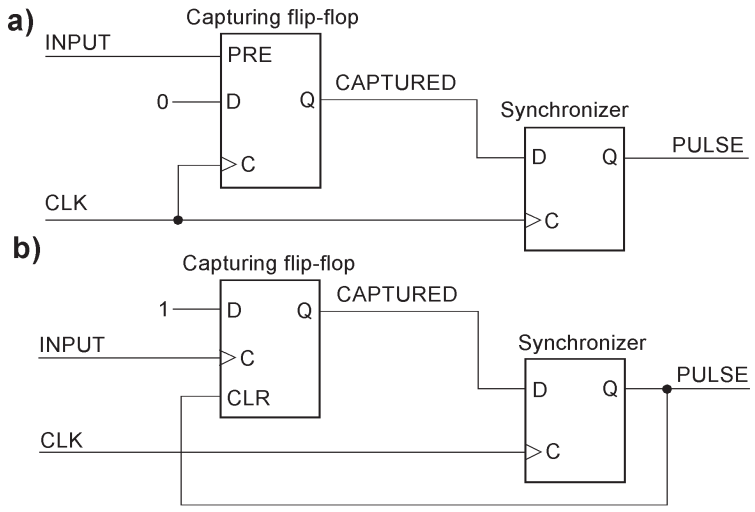


**Figure 1** Synchronizing narrow pulses (1). The circuits differ in their behavior when the asynchronous input pulse is wider than a clock cycle (a) or when a new pulse occurs while the synchronizer's output is still active (b).

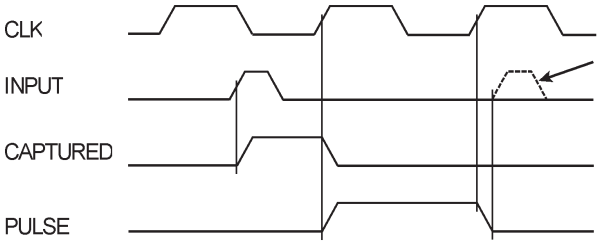


**Figure 2** Synchronizing narrow pulses (1). The pulses occur widely apart from one another. The next pulse must not appear before the synchronizer flip-flop has become inactive again (as the arrow points to). Then, the circuits of Figure 1a and 1b behave the same way.

If the pulses are wider than suggested in Figure 1 or follow more closely one another, the circuits will behave differently, as shown in Table 1 and Figures 3 and 4.

| **Characteristics of the input pulses** | **Figure 1a** | **Figure 1b** |
|---|---|---|
| A new pulse occurs when the output is still active | The output pulse will be extended by a further clock cycle | This pulse will be ignored |
| The pulse is longer than a clock cycle | The output pulse will be extended by an appropriate number of clock cycles | The width of the output pulse is always one clock cycle |

**Table 1** Differences in the behavior of both circuits.

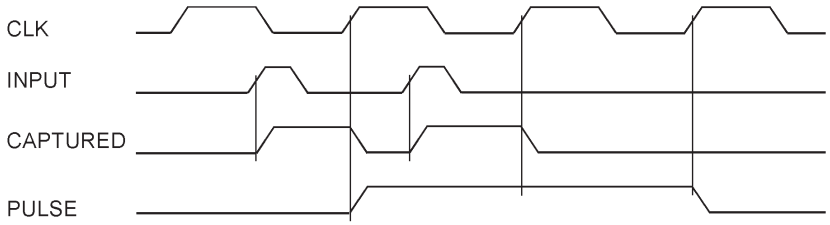


**Figure 3** Synchronizing narrow pulses (2). The next pulse occurs while the synchronization flip-flop is still active. The level of the input signal takes precedence. Thus, the circuit of Figure 1a will extend the output pulse by another clock cycle.

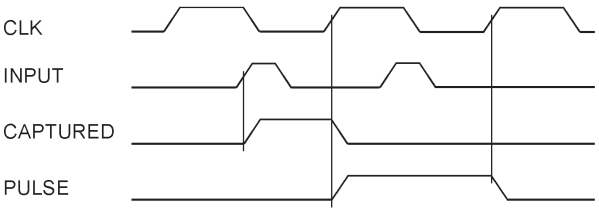


**Figure 4** Synchronizing narrow pulses (3). The next pulse occurs while the synchronization flip-flop is still active. The clear signal takes precedence. Thus, the circuit of Figure 1b will ignore the second input pulse.

**Metastability analysis**

Flip-flops are prone to metastable states if asynchronous preset and clear signals overlap with the clock edge. The preset signal PRE of the capturing flip-flop in Figure 1a always takes precedence over the clock. A metastability problem could only occur if the clock goes active while the input becomes inactive (Figure 5). Then, the input may win (that is, the flip-flop sees PRE still as active during the clock edge and hence does not memorize the Low at the D input). Thus, the output remains asserted (Figure 5a). Alternatively, the clock may win and the output becomes deasserted (Figure 5b), or there is no winner and the flip-flop goes through a metastable state (Figure 5c).

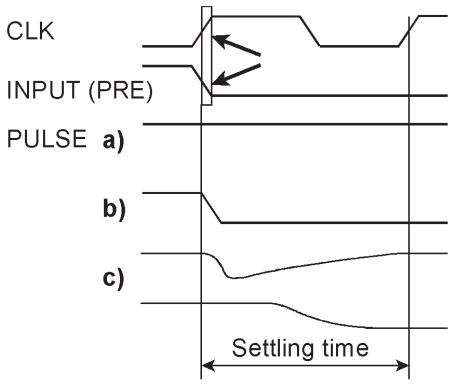


**Figure 5** Metastability problems (1). The coincidence of edges that the arrows point to may occur in the circuit of Figure 1a. PRE relates to the preset input of the capturing flip-flop.

In Figure 1b, the clear signal takes precedence over the asynchronous input acting as the flip-flop's clock. A metastability problem could only occur if the input goes active while the synchronizer's output becomes inactive (Figure 6). A new pulse occurs in the moment the synchronizer has cleared the preceding pulse memorized by the capturing flip-flop. Therefore, to understand the problem, we must illustrate the history beginning with the previous pulse. Thus, Figure 6 is somewhat more elaborate than Figure 5.

If CLR wins, the capturing flip-flop remains deasserted (Figure 6a). If C wins (that is, the asynchronous pulse input), the capturing flip-flop will switch on again (Figure 6b), causing the output to remain asserted during the next clock cycle. If there is no winner, the capturing flip-flop goes through a metastable state (Figure 6c).

In both circuits, one of the signals whose overlapping could cause metastable states is asserted or deasserted by the synchronizer and hence by a clock edge. Therefore, the settling time always equals one complete clock cycle. So we can assume that the synchronizer will see a settled input level when the next clock cycle begins. As a consequence of the metastable state, it could be Low or High, respectively. Accordingly, the output pulse will end or remain active during the next clock cycle (Figure 5c and 6c).

Both circuits will perform well if the width of the input pulses surpasses the flip-flop's pulse width data sheet parameter and the gaps between the pulses are wider than a clock period.

Moreover, the clock cycle provides the settling time to prevent failures caused by metastable states. So it should be chosen at least as wide as necessary to ensure the specified MTBF. When not specified, choose a reasonable magnitude, perhaps some centuries. The problem has been discussed in the preceding article [3].

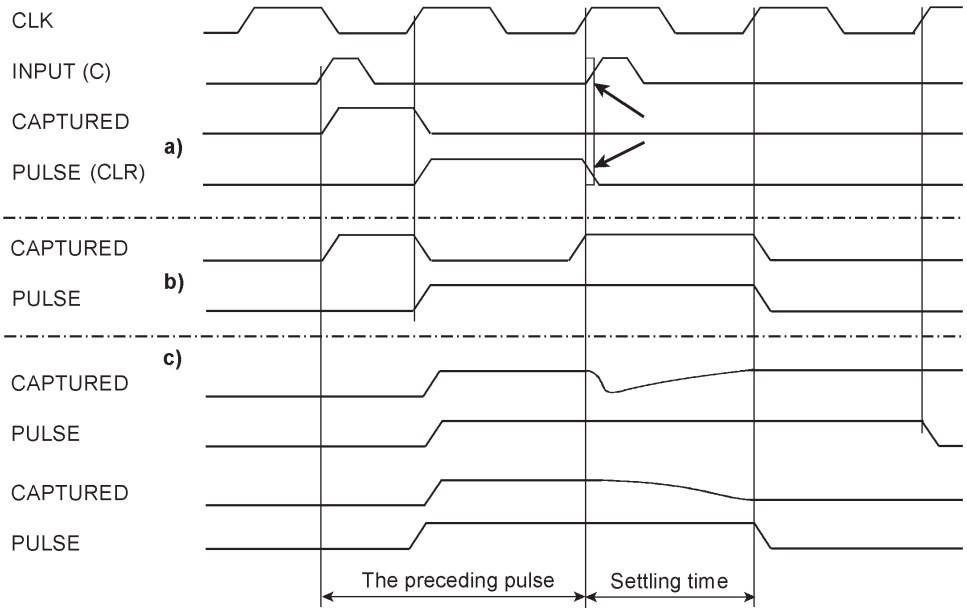


**Figure 6** Metastability problems (2). The coincidence of edges that the arrows point to may occur in the circuit of Figure 1b. C and CLR relate to the clock and clear inputs of the capturing flip-flop.

## Synchronize pulses continuously

Figure 7 shows a circuit that avoids the drawback of the circuit shown in Figure 1b, that the pulse arriving while the PULSE output is asserted will be lost. Therefore, two circuits according to Figure 1b are employed. The second circuit will take over while the first has memorized a pulse or is cleared. Two consecutive pulses are signaled by the PULSE output active for two clock cycles (Figure 8).

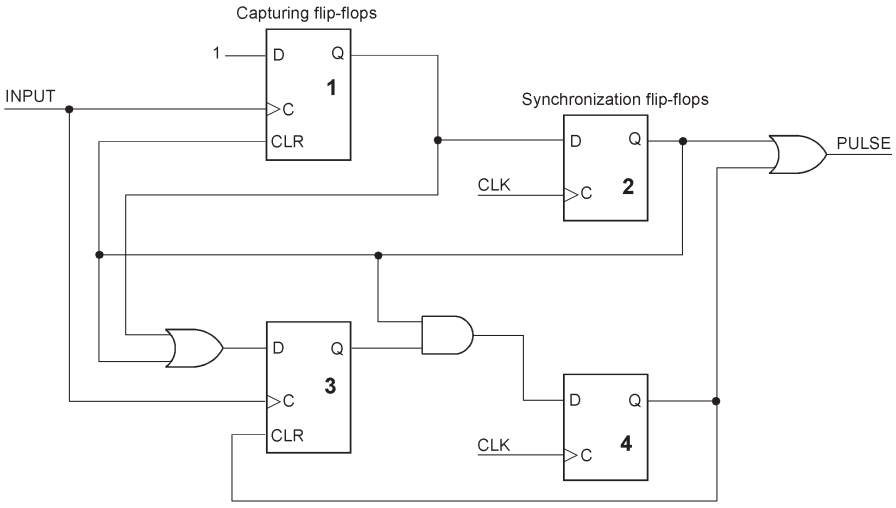

**Figure 7** Continuous synchronization with two capturing and synchronizing circuits working in a teeter-totter operation. The first input pulse is memorized by flip-flop 1 and synchronized by flip-flop 2. During this clock cycle, flip-flop 1 is reset and cannot capture a second pulse. Instead, the flip-flops 3 and 4 take over. In the clock cycle that resets flip-flop 3, flip-flop 1 is ready again to capture the next pulse, and so on.

The circuit will work properly if no more than two input pulses occur within an interval two clock cycles wide, even when the second pulse arrives before the capturing-flip-flop 1 is cleared (Figure 9). The OR gate in front of the capturing flip-flop 3 ensures that a second pulse will always be caught when the first capturing and synchronizing circuit (consisting of the flip-flops 1 and 2) is busy, that is, if the first pulse has been captured or is output.

The circuit converts sequences of narrow pulses into an NRZ (Non-Return-to-Zero) representation. Two consecutive narrow pulses will cause an output pulse two clock cycles wide. More consecutive pulses will produce an appropriately wider output pulse, as shown in Figure 10.

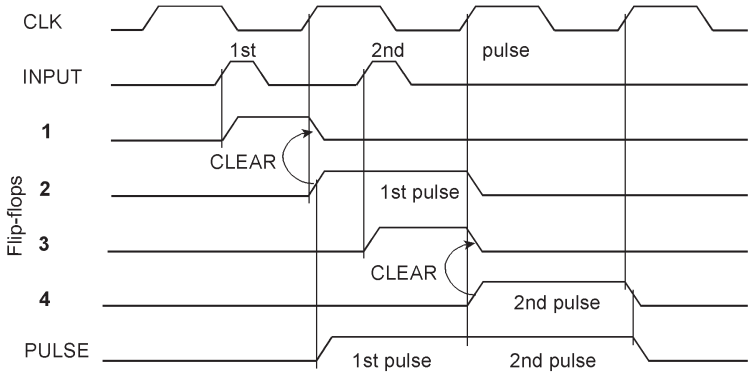


**Figure 8** Example waveforms explaining the circuit's operation. A second input pulse arrives when the first capturing flip-flop (1) is still cleared by the synchronization flip-flop (2). The newly arrived pulse will be memorized by flip-flop (2) and synchronized by flip-flop (3).

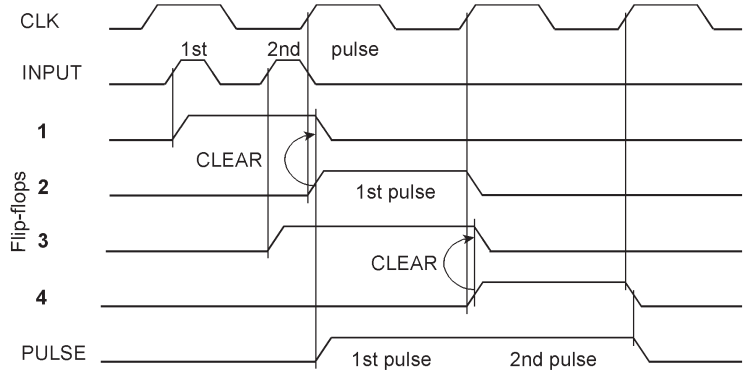


**Figure 9** This timing diagram shows two input pulses arriving close together.

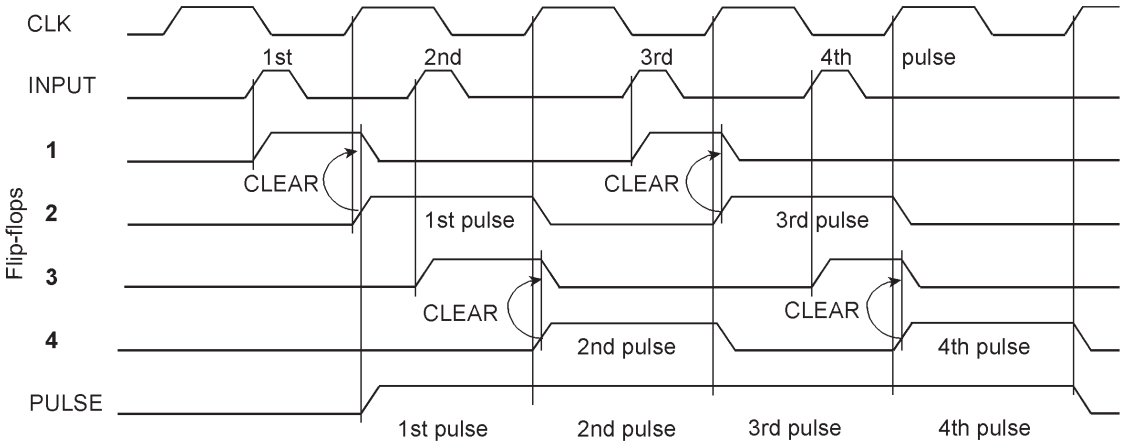


**Figure 10** Converting pulses into an NRZ representation. Here, four arriving pulses cause an output pulse four clock cycles long.

The circuits in Figures 1 and 7 could be implemented by tiny logic or in CPLDs. They are adequate for asynchronous pulses down to tens of nanoseconds and frequencies up to some MHz. When the design task is more demanding, I recommend sampling and synchronizing the pulses by an appropriate high-frequency clock and solving the particular application task of memorizing, deglichting, and the like by a downstream finite state machine (FSM), as shown in Figure 11.

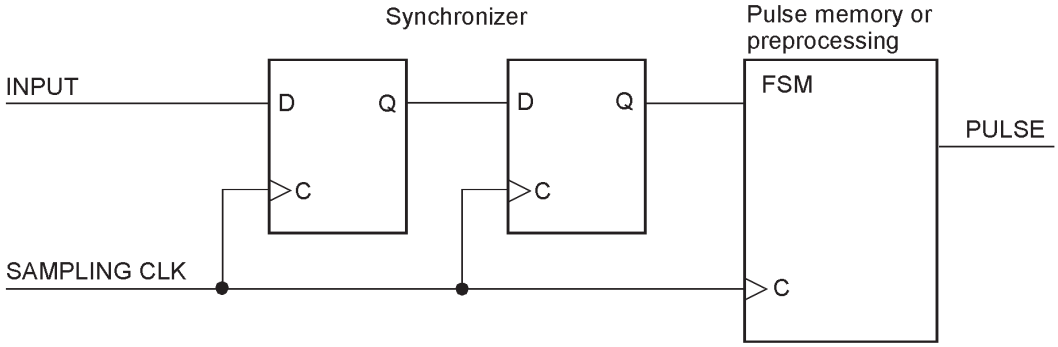


**Figure 11** Solving pulse memorizing and preprocessing tasks (like, for example, glitch detection and debouncing) by synchronous finite state machines (FSMs). The arriving pulses are sampled and synchronized. Here, a two-stage synchronizer is shown. The application problem can be solved by state machine design as taught in the textbooks and supported by hardware description languages (HDLs) and integrated development systems.

## Detecting complete pulses

A complete pulse has two edges. They can be detected by two capturing flip-flops whose outputs are ANDed together (Figure 12a). To detect a particular kind of pulse (active-High or active-Low), two flip-flops are connected in series, like a shift register (Figure 12b). It is the most basic kind of sequence detector.

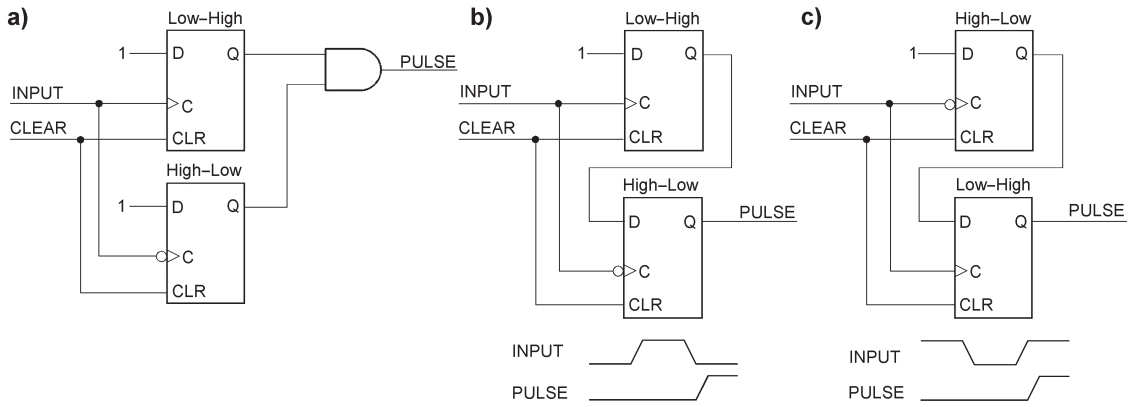


**Figure 12** Detecting complete pulses. The circuit (a) detects both edges regardless of how they follow each other. The circuits (b) and (c) detect pulses beginning with a Low-to-High or High-to-Low edge, respectively.

## Detecting edges

An edge detector circuit emits a pulse if the input level changes to the opposite (from Low to High or vice versa). Edges could be detected by connecting the signal to be observed to the clock inputs of edge-triggered flip-flops, as depicted in Figure 12. Those flip-flops, however, must be appropriately cleared to arm them again to capture the next edge.

An alternative approach is to delay the signal and compare the delayed level with the current one (Figures 13 and 14). The limiting parameters are the minimum pulse width $t_W$ in the pulse train and the minimum pulse width $t_{WE}$ that the edge detector circuits expect. $t_{WE}$ approximately equals the delay time $\Delta t$. The delay must be shorter than the pulse ( $\Delta t < t_W$); otherwise, the circuit will not work (as clearly visible in Figure 13c).

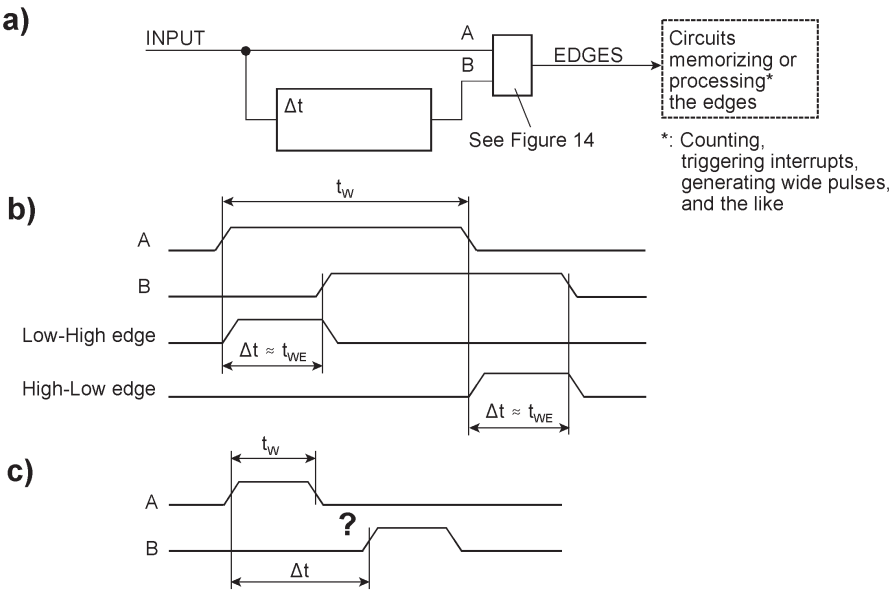


**Figure 13** Edge detection by delaying. (a) depicts the principal circuit, (b) how both edges are detected. The combinational functions of A and B are shown in Figure 14. (c) illustrates that the circuit will not work when the arriving and the delayed pulse do not overlap.

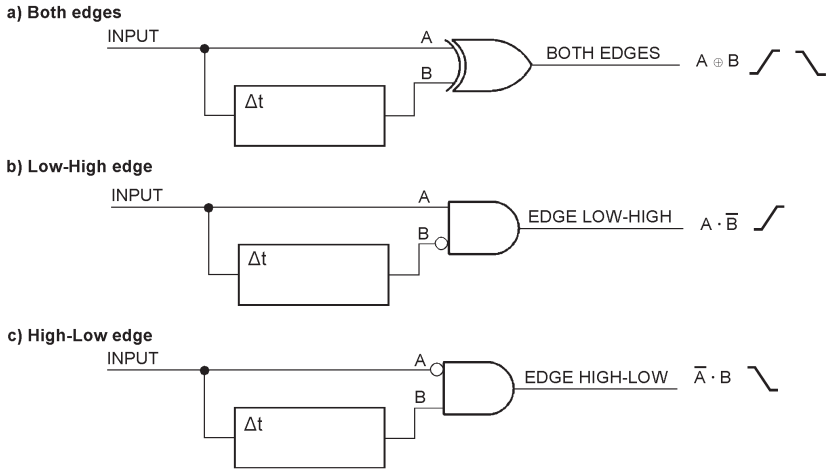


**Figure 14** Basic edge-detection circuits.

The output signals are typically pulses slightly shorter than the delay time Δt. Because Δt must be beyond the minimum width of the input pulses, the output pulses will be conspicuously narrow. In many application circuits, they must be stretched or memorized.

There are various principles to delay a signal. It can be done by analog or digital components. When the arriving pulses are particularly narrow (think of micro- or even nanoseconds), it might be appropriate to implement the edge detection in the asynchronous domain, perhaps by analog circuits, and provide a fast pulse-capturing or single-shot circuit downstream. In a synchronous clock domain, the most straightforward digital delay component is the D-type flip-flop (Figure 15).

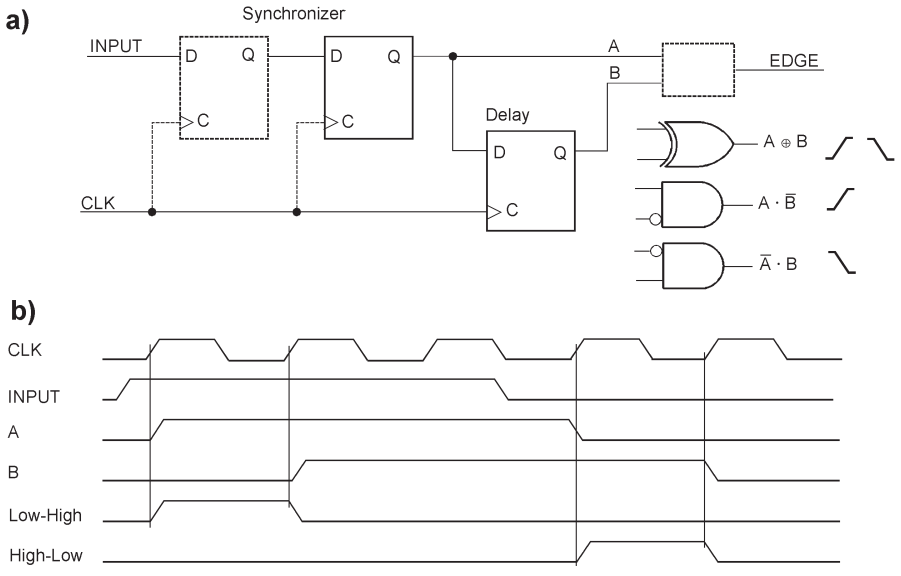


**Figure 15** Synchronous edge detection. A D-type flip-flop delays the synchronous pulses. The edges are signaled by output pulses being a single clock cycle wide. If wider output pulses are needed, several delaying flip-flops could be connected in series (that is, as a shift register), provided the pulses of the input pulse train are wide enough (that is, wider than the delay).

## Pulse stretching and recovery

An alternative approach for dealing with narrow pulses is to stretch them, that is, to make them wider instead of memorizing them. A toggle flip-flop converts arbitrarily wide and even extremely narrow pulses to symmetric pulses that retain their level until the next input pulse arrives. Basically, it is a 2:1 frequency division (Figure 16). Each edge of the 2:1 pulse train signals an original pulse. Hence, the original pulse train can be reconstructed by detecting both edges (Figure 17).

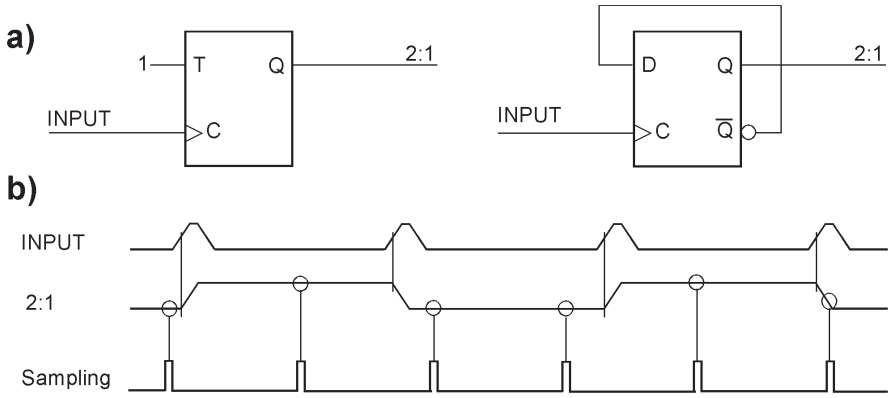


**Figure 16** Stretching pulses by a 2:1 frequency division. (a) shows a genuine T flip-flop and a D flip-flop turned into such. As shown in (b), narrow pulses are converted into symmetrical wide pulses that can be sampled reliably.

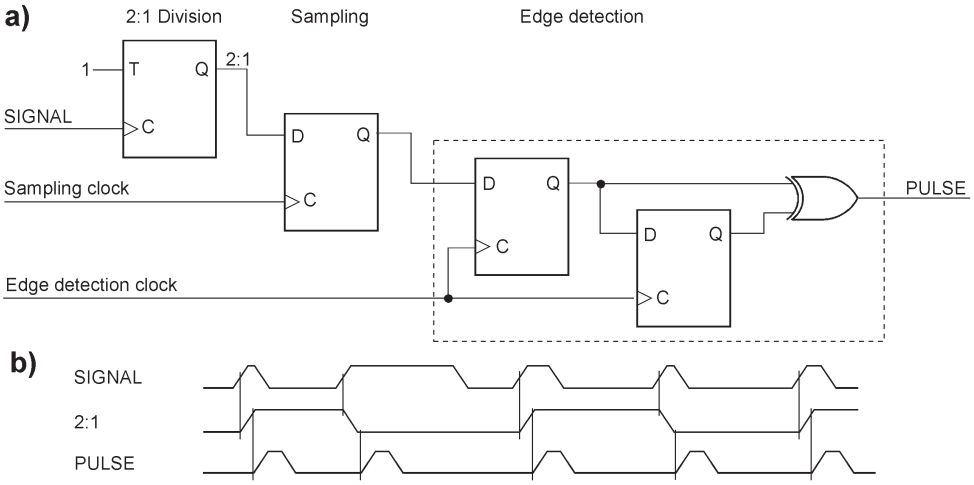


**Figure 17** Reconstruction of the original pulse train by detecting edges. Here, it is done by a synchronous edge detector (a). Its clock determines the width of the output pulses. The delay must be shorter than the narrowest pulse. Therefore, the edge detection clock frequency must be higher than the sampling clock frequency. To ensure this, the sampling clock could be, for example, derived from the edge detector's clock by frequency division. (b) shows example waveforms. A practical application example is crossing clock domains. The SIGNAL comes from another clock domain. This pulse train is injected into the target clock domain, to which the sampling and edge detection clocks belong.

## Detecting sequences

An occasional task is to detect whether particular signals switch in a specified sequence order. For example, assume signal lines A, B, C, and so on. Pulses may occur, and we want to recognize whether they appear in a certain order. The circuit should activate an output signal when, for example, a sequence A – C – D... occurs, but not with every other sequence. This can be done by a chain of D-type flip-flops whose clock inputs are connected to the individual signals according to the sequence to be detected. The last flip-flop in the chain will only become active if all signals have switched in the stipulated sequence.Figure 18 illustrates the principle by an example easily to understand. By the way, the circuit is simple but dangerous. As brain teasers: (1) How can you crack such a lock? (2) How can you detect and prevent cracking?

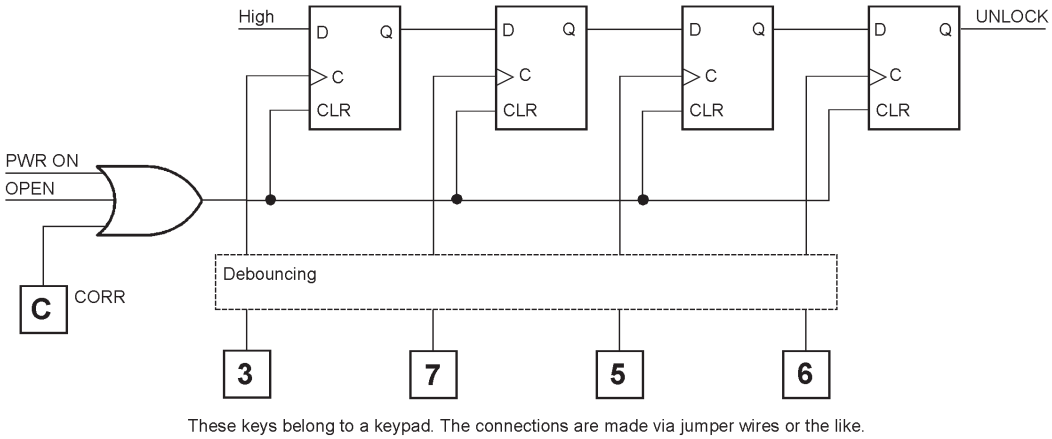


**Figure 18** Sequence detection shown by the example of a straightforward code lock. The UNLOCK signal actuates the door opener. The flip-flops are cleared during power-on, when the door has been opened, or when the CORR key is depressed. The number keys are part of a keypad (with the keys 0...9; arrangement similar to a telephone or calculator). The key signals must be bounce-free. For the UNLOCK output signal to become active, the number keys 3, 7, 5, and 6 must be pressed in precisely this order. If the door has been opened, the OPEN signal activated by a door contact resets the code lock.

## Capturing pulses synchronously

CPLDs and FPGAs can operate at extreme clock frequencies. If every pulse, no matter how narrow, is indispensable and cannot be ignored, the sampling frequency must be high enough. All asynchronous signals are first synchronized with a sufficiently high clock frequency and captured or processed using state machines (synchronous preprocessing). CPLDs and FPGAs can operate with clock frequencies between 50 MHz and much more than 200 MHz. Relying on our rule of thumb (sampling frequency = four times the pulse repetition frequency of the narrowest symmetric pulses), we may capture and process pulses that are between 40 ns and 10 ns wide (practically, somewhat less because of the flip-flop's setup time). What to do with pulses being still narrower depends on the application. They may, for example, be suppressed or stretched accordingly by analog filtering or preprocessing.

### Basic capturing tasks

Sampling and synchronization are the first stage of capturing. No pulse will escape if we sample fast enough. The question is how wide the synchronized PULSE output signal should be. The

width could be, for example, approximately equal to that of the input pulse. For this, the synchronizer alone suffices, behaving similarly to the circuit of Figure 1a. If we want a synchronized pulse only one clock cycle wide, regardless of the input pulse width, we must connect the synchronizer to a single-shot generator.

**Synchronous pulse memories**

Synchronous pulse memories are finite state machines (FSMs) that, when a particular signal level is sampled, enter a state in which they remain until a clear signal becomes active. All signals we are concerned with here are clock-synchronous. Figures 19 and 20 depict synchronous pulse memories differing in their precedence behavior.

The most straightforward FSMs have only two states. When the arriving pulses and clearing of the pulse memory do not overlap, we can get by with an IDLE state, when nothing is to memorize, and a PULSE state, if the input has become active (Figure 19). When the input pulse has disappeared, the feedback will cause the PULSE state to be retained until CLEAR becomes active and disconnects the feedback. Thus, the pulse memory will fall back to its IDLE state. If, however, CLEAR is asserted while INPUT is active, the latter will take precedence, causing the FSM to remain in the PULSE state.

To circumvent this behavior, the pulse memory FSM of Figure 20 will accept a pulse only when both signals, INPUT and CLEAR, have been inactive together previously. For this, a third state has been added to wait for the said condition. The IDLE state will only be taken again after the input pulse and the CLEAR signal have both become inactive. Afterwards, the FSM is armed again to catch the next arriving pulse.

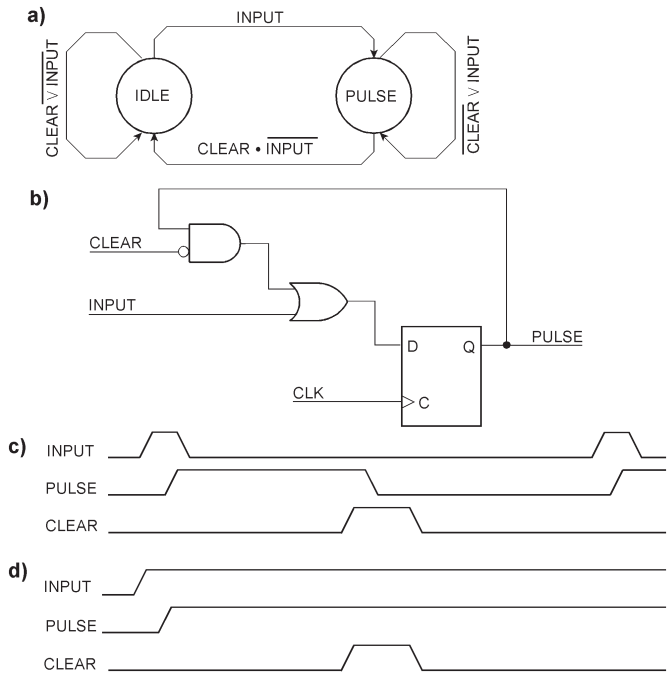


**Figure 19** This synchronous pulse memory detects the signal level. The pulse input takes precedence over CLEAR. a) state diagram; b) circuit example. c) A short pulse is memorized and then cleared. d) If the pulse input is permanently active, CLEAR does not affect the PULSE state. An illustrative application example is the capturing of an error signal. If PULSE diminishes after asserting CLEAR, it is a transient error; if not, it is a permanent fault.

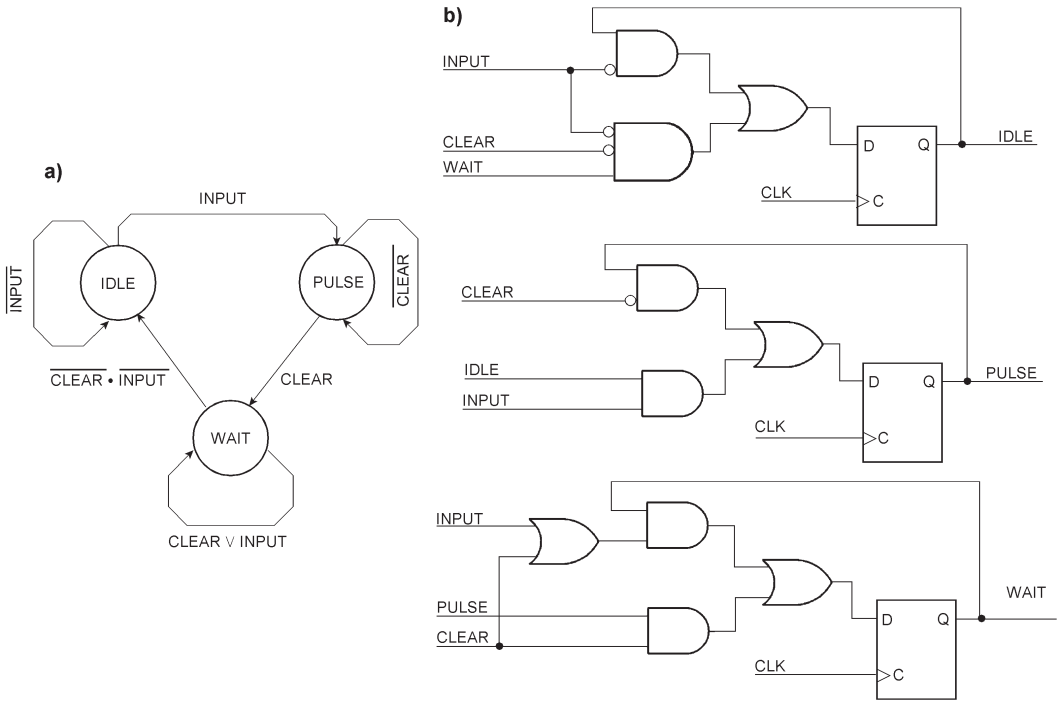


**Figure 20** This synchronous pulse memory detects the pulse's Low-to-High edge. Here, CLEAR takes precedence over the pulse input. After the CLEAR signal has been deasserted, the circuit will wait until the pulse input becomes inactive (Low) too. a) state diagram; b) circuit example. The state encoding is one-hot enable (OHE). On power-on, IDLE must be set; PULSE and WAIT cleared.

## Example single-shot generators

When an input pulse arrives, the single-shot generator emits an output pulse one clock cycle wide. Single-shot pulse generators could be dubbed digital monostable multivibrators, too. They can be designed as finite state machines(FSMs), their behavior described by state transition diagrams. Appropriate state machines must provide for wait states in which they wait for the input signal to become inactive (Figure 21).

When the input signal is asynchronous, it must first be synchronized. When it could be affected by glitches or bouncing, it must be debounced, to boot. According to Figure 21a, the FSM will always go through the wait state, even if the input signal has already become inactive. Figure 21c shows how, in this case, the wait state can be skipped.

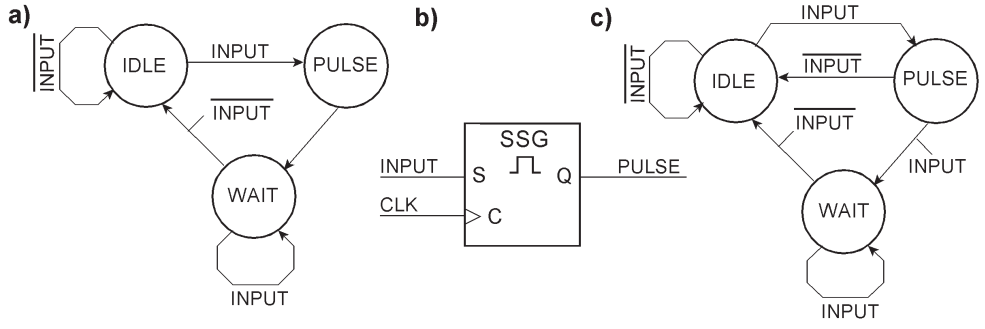

**Figure 21** Principles of synchronous single-shot pulse generation. a) state diagram; b) circuit symbol; c) modification for skipping the WAIT state if the input signal becomes inactive while still in the PULSE state. Each state transition requires one clock cycle.

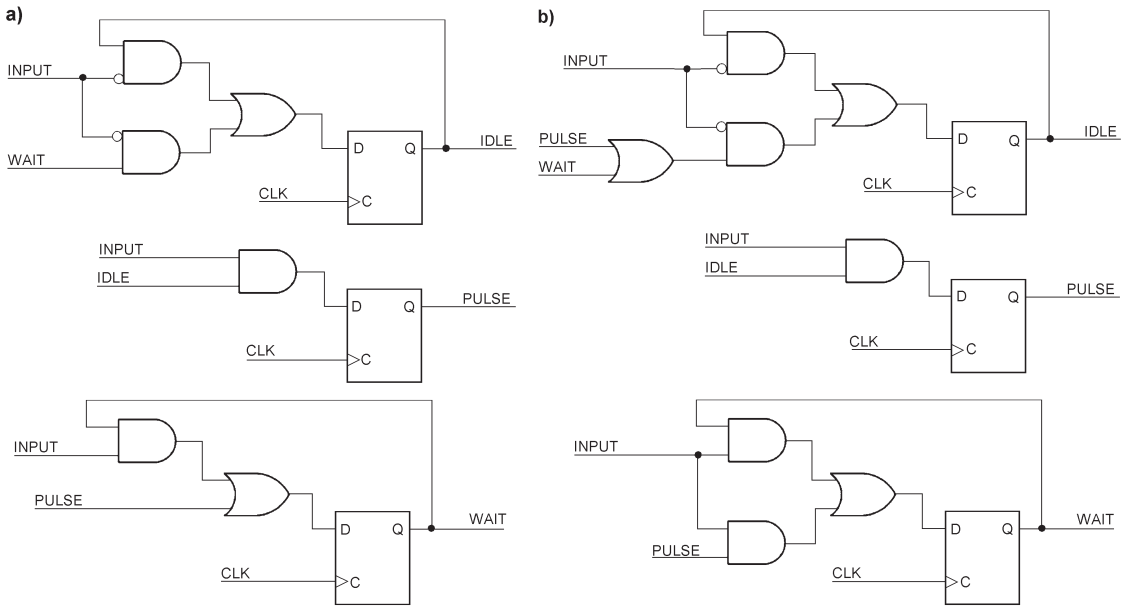


**Figure 22** Single-shot pulse generation by OHE-encoded FSMs. On power-on, IDLE must be set; PULSE and WAIT cleared. a) implements the state diagram of Figure 21a; b) that of Figure 21c.

# Glitches

In some applications, the input pulses must have a certain minimum width. Conspicuously narrow pulses are seen as spikes, glitches, and other types of noise. Only pulses distinctively wider are considered acceptable. There are two principal tasks: suppressing the spikes or glitches (deglitching, debouncing) and filtering them out, for example, to be counted, recorded, and analyzed.

There are two principal definitions of what constitutes a glitch. According to the "asynchronous" definition, a glitch is any pulse narrower than a specified (short) time. The "synchronous" definition relates to a specified clock cycle. A glitch is a complete pulse (more generally, any waveform showing at least two edges) during a clock cycle. Oscilloscopes see asynchronous, logic analyzers see synchronous glitches.

**Detecting asynchronous glitches in a synchronous enviroment**

An entire clock cycle must be monitored to see whether the input signal has at least two edges in this interval. Figure 23 depicts the problem.

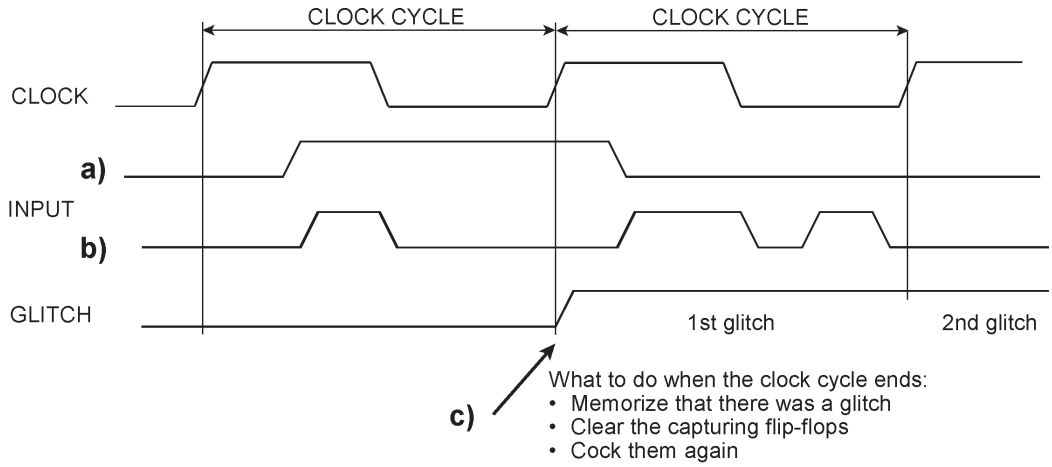


**Figure 23** Detecting asynchronous glitches occurring within clock cycles. The input pulse (a) shows only one edge in each clock cycle. Hence, it is not a glitch. In contrast, the pulse train (b) carries glitches during two adjacent clock cycles. The arrow (c) points to the end of a clock cycle and the beginning of the next. In this moment, the stipulated activities of meorizing, clearing, and cocking again are to be carried out.

Detecting edges is easy. It could be done by edge-triggered flip-flops. However, the clock cycles follow one another without any gaps. Figure 23c shows what to do at the moment of the transition from one clock cycle to the next. Because we use the clock inputs of our capturing flip-flops to detect glitches, synchronous operation is out of the question. What we cannot do is resort to advanced transistor-level design, like the manufacturers of digital storage oscilloscopes and logic analyzers. Instead, we can only think of straightforward circuits built by readily available components. Figure 24 illustrates a solution. Two detection circuits, each consisting of two pulse memories, are used alternately in successive clock cycles A and B. During clock cycle A, the lower detection circuit is active, and the upper one is held in reset. During clock cycle B, the modes of operation are reversed. When changing from one clock cycle to another, the output signal of the respective detection circuit is transferred to a synchronization flip-flop.

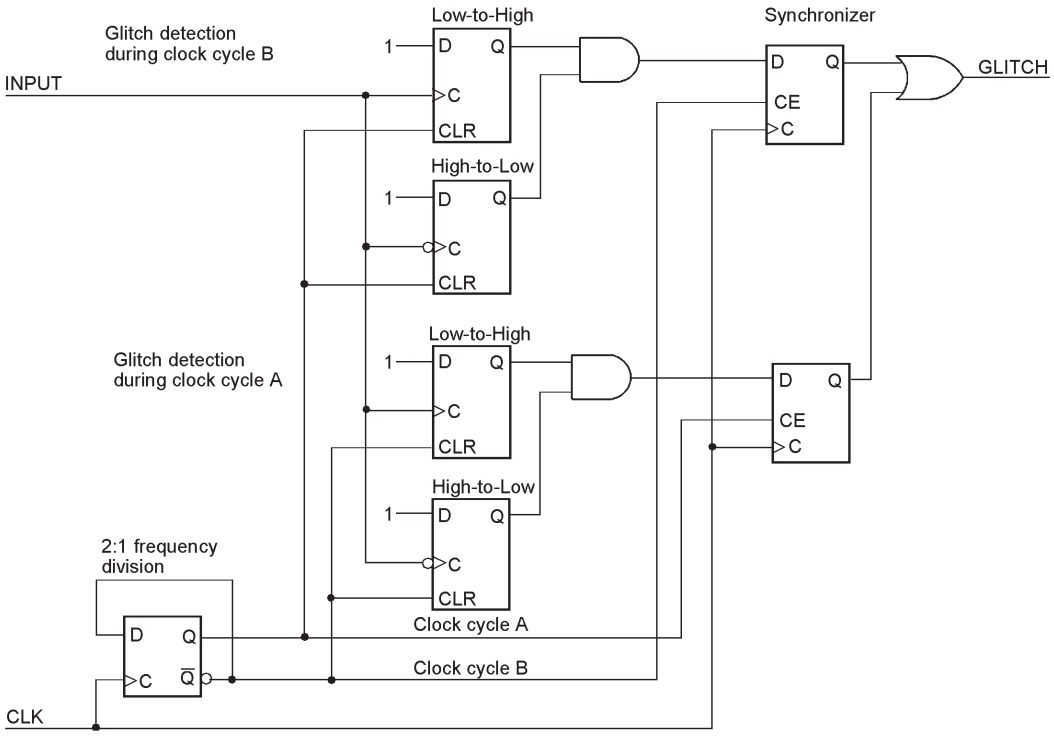


**Figure 24** Seamless glitch detection in a synchronous environment. Two circuits in teeter-totter operation ensure that no glitch escapes capturing.

## A legacy debouncing synchronizer

The circuit in Figure 25 only emits a synchronized pulse if the input pulse is wide enough. When a narrow pulse does not coincide with a Low-to-High clock edge, nothing happens. If the pulse and a clock edge meet, the pulse memory flip-flop will be set. If the pulse becomes inactive again, the pulse memory flip-flop is cleared immediately. A captured pulse is only synchronized if the pulse memory flip-flop has not been cleared before the next Low-to-High edge of the clock.

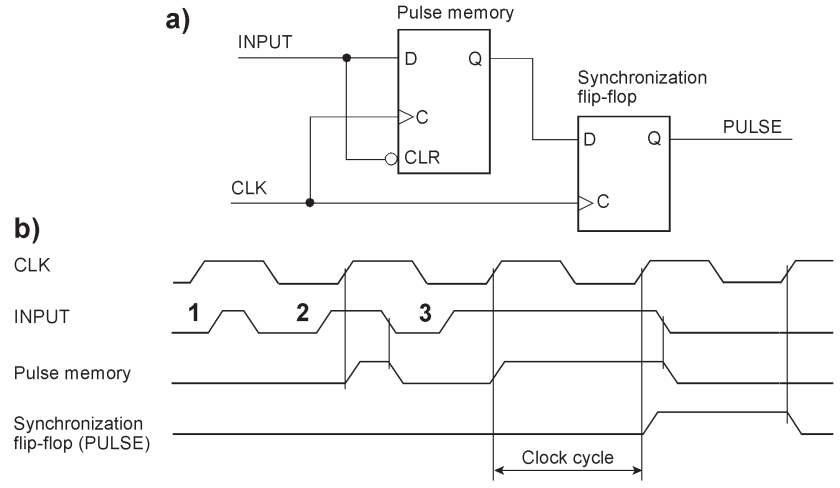


**Figure 25** This venerable circuit synchronizes only pulses wide enough (deglitching, debouncing). a) the circuit (according to [A3]); b) an example waveform diagram.

The too-narrow pulse (1) does not coincide with a low-to-high edge of the clock. Hence, it will not be synchronized. The too-narrow pulse (2) is captured but will clear the pulse memory flip-flop during the current clock cycle. Hence, it will not be synchronized. In contrast, the pulse (3) is wide enough. It is held in the pulse memory flip-flop until the next Low-to-High edge of the clock and thus synchronized. If the pulse is slightly wider than one clock cycle, it may be synchronized or not. To be synchronized, it must remain active until the next Low-to-High clock edge. If the pulse is somewhat wider than two clock cycles, it will be reliably synchronized.

The circuit is only usable if it does not matter whether pulses wider than one but narrower than two clock periods are being synchronized or not. A typical application example is contact debouncing, where the clock can be generated even by software (we speak of cycles in the magnitude of some milliseconds here).

The circuit in Figure 25 we have called venerable because of its age. Due to its age, however, there is also a caveat. Decades-long experience has shown that it works well, for example, with the SN7474 D-type flip-flop. However, when implemented by up-to-date components, it may not work faultlessly. It could even be possible that the integrated development environment (IDE) refuses to synthesize it. The cause is that the INPUT signal acts upon the synchronous D as well as the asynchronous CLR input of the pulse memory flip-flop.

**Synchronous deglitching and glitch detection**

The following circuits are based on principles of synchronous operation to suppress or detect glitches. Essentially, they are finite state machines (FSMs). Hence, their operation can be described by state transition diagrams.

If the pulse is narrower than a clock cycle, it is considered a spike or glitch; if wider, it is considered acceptable. Therefore, choose the clock frequency according to the pulse width that makes the difference between a glitch and an acceptable pulse.

In a completely synchronous circuit, the signals to be evaluated are already synchronized. We assume a synchronization clock cycle corresponding to the maximum width that is considered to be a glitch. Then, a glitch corresponds to a one between two zeros or a zero between two ones. Consequently, acceptable pulses consist of at least two consecutive zeros or ones.

Successive snippets of a pulse train can be sequentially memorized using flip-flops connected in series. In the simplest case, three clock periods must be considered: the signal level synchronized in the current cycle and those of the two preceding cycles. To analyze the pulse patterns that might occur, we set up a truth table that contains all eight possible signal combinations (Figure 26).

a)

| C1 C2 C3 | IN | D | Q | PULSE | GLITCH | State |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0,0 |
| | 1 | 0 | 0 | 0 | 0 | 0,0 |
| | 0 | 1 | 0 | 0 | 1 | 1,0 |
| | 0 | 0 | 0 | 0 | 0 | 0,0 |
| | 1 | 1 | 1 | 1 | 0 | 1,1 |
| | 0 | 1 | 1 | 1 | 0 | 1,1 |
| | 1 | 0 | 1 | 1 | 1 | 0,1 |
| | 1 | 1 | 0 | 1 | 0 | 1,1 |

t →

$\text{GLITCH} = \overline{\text{IN}} \cdot D \cdot \overline{Q} \vee \text{IN} \cdot \overline{D} \cdot Q$

$\text{PULSE} = \text{IN} \cdot D \vee \text{IN} \cdot Q \vee D \cdot Q$

b)

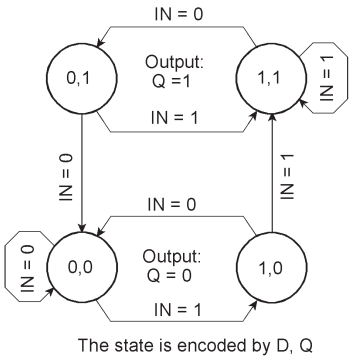


**Figure 26** Synchronous glitch filtering and detection (based on [A4]). a) truth table, b) state diagram of the glitch filter's finite state machine (FSM). The Boolean equations relate to the glitch filter circuit of Figure 27 and the glitch detector of Figure 28.

C1, C2, and C3 are the three successive clock cycles we want to observe. The truth table contains all eight possible pulse patterns. We begin at a Low level. Then a High glitch emerges, becomes visible, and diminishes again. The second half of the table begins at a High level. Then a Low glitch emerges, becomes visible, and diminishes again.

The FSM's state is encoded by the flip-flops D and Q. In each clock cycle, the output depends on the input and the state. The input (IN) comes from the synchronizer. Because it is already clock-synchronous, we need only two state flip-flops. To suppress glitches, the circuit in Figure 27 will emit a zero if two or more zeros or a one if two or more ones follow each other. It will not change its output if a single one is interspersed between two zeros or a single zero between two ones. Precisely these conditions define a glitch. They are detected by the circuit in Figure 28.

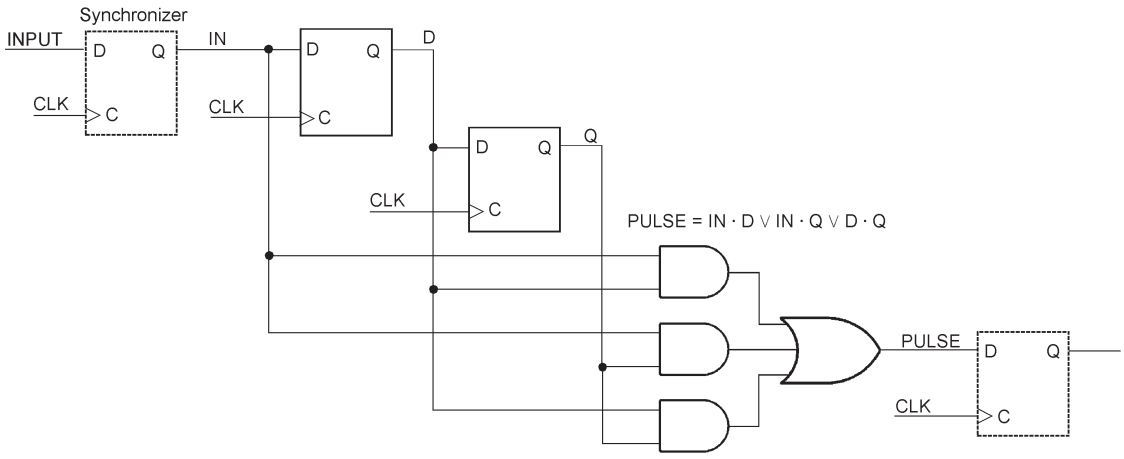


**Figure 27** A glitch filter or de-glitching circuit based on the truth table in Figure 26. It outputs a pulse train free of glitches.

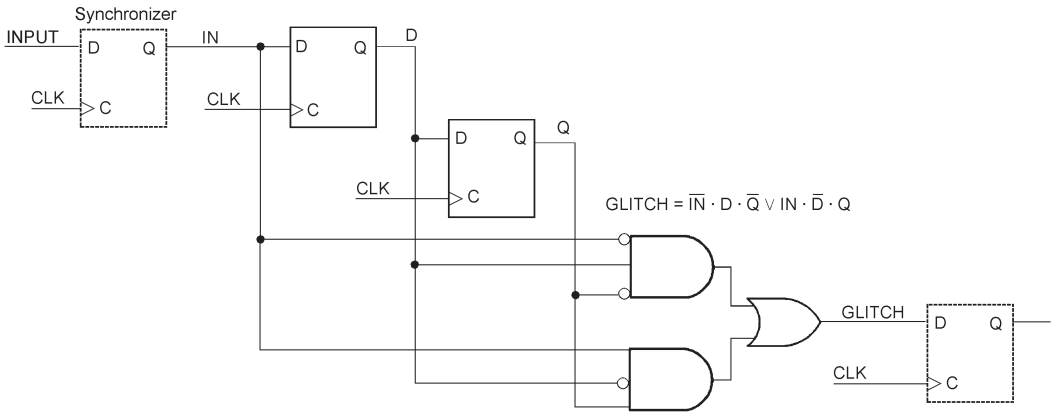


**Figure 28** A glitch detector circuit based on the truth table in Figure 26. It outputs pulses representing glitches. Thus, they could be memorized, highlighted, counted, and the like.

Figure 29 shows a glitch filter consisting of a shift register, a combinational circuit detecting particular bit patterns, and an output flip-flop. The pulse train is fed into a shift register. The output flip-flop may change its state only if the shift register contains three ones or zeros, thus suppressing all other bit patterns mentioned in Figure 26. The shift register's output is then memorized in the output flip-flop.

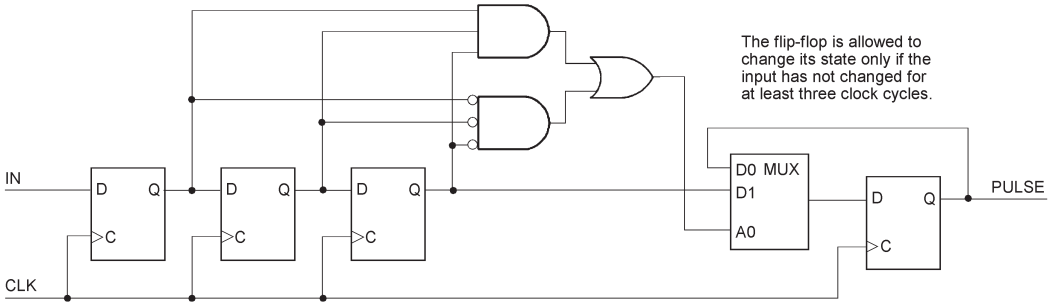


**Figure 29** Another glitch filter or de-glitching circuit shows an obvious, easy-to-understand principle of operation. It is also obvious that it could be used to suppress wider glitches by simply making the shift register longer and the AND gates wider.

## A crucial difference between asynchronous and synchronous glitch filtering

Remember that the circuits downstream cannot see how the signal behaves between the sampling moments. Figure 30 shows a somewhat subtle waveform. Asynchronous and synchronous circuits will react differently. According to Figure 25 (asynchronous), no pulse will appear on the output. According to Figure 29 (synchronous), a pulse will be output because three consecutive clock edges have sampled the same signal level. Therefore, analyze your application requirements carefully and be wise when selecting an appropriate circuit.

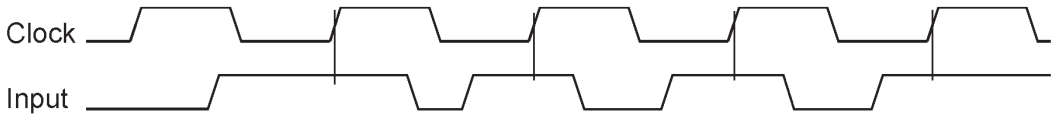


**Figure 30** These waveforms show the difference between asynchronous and synchronous de-glitching.

**Supplementary references:**
This addendum refers to the References and Resources of the printed article. The following additional references concern the reset problems mentioned at the beginning and two legacy primary sources of some deglitching and glitch-detecting circuits.

[A1] RES-50001: Asynchronous Reset Is Not Synchronized.
https://www.intel.com/content/www/us/en/programmable/quartushelp/current/index.htm#da_rules/res_50001.htm

[A2] RES-50002: Asynchronous Reset is Insufficiently Synchronized.
https://www.intel.com/content/www/us/en/programmable/quartushelp/20.1/index.htm#da_rules/res_50002.htm

[A3] Tietze, U.; Schenk, Ch.: Halbleiter-Schaltungstechnik. Springer 1991.

[A4] Jakubaitis, E. A.: Asynchronous logical automata (in Russian). Riga 1966.