

time, the output may be unpredictable and the time for the output to settle to a good logic level may be unbounded. Properly designed synchronous circuits guarantee the data is stable during the aperture. However, many interesting systems must interface with data coming from sources that are not synchronized to the same clock. For example, the user can press a key at any time and data coming over a network can be aligned with a clock of differing phase or frequency.

A *synchronizer* is a circuit that accepts an input that can change at arbitrary times and produces an output aligned to the synchronizer's clock. Because the input can change during the synchronizer's aperture, the synchronizer has a nonzero probability of producing a *metastable* output. This section first examines the response of a latch to an analog voltage that can change near the sampling clock edge. The latch can enter a metastable state for some amount of time that is unbounded, although the probability of remaining metastable drops off exponentially with time. Therefore, you can build a simple synchronizer by sampling a signal, waiting until the probability of metastability is acceptably low, then sampling again. In certain circumstances, the relationship of the data and clock timing is more predictable, permitting more reliable synchronizers.

### 7.6.1 Metastability

A latch is a bistable device; i.e., it has two stable states (0 and 1). Under the right conditions, that latch can enter a metastable state in which the output is at an indeterminate level between 0 and 1. For example, Figure 7.79 shows a simple model for a static latch consisting of two switches (probably transmission gates in practice) and two inverters. While the latch is transparent, the sample switch is closed and the hold switch open (Figure 7.79(a)). When the latch goes opaque, the sample switch opens and the hold switch closes (Figure 7.79(b)). Figure 7.79(c) shows the DC transfer characteristics of the two inverters. Because  $A = B$  when the latch is opaque, the stable states are  $A = B = 0$  and  $A = B = V_{DD}$ . The metastable state is  $A = B = V_m$ , where  $V_m$  is not a legal logic level. This point is called metastable because the voltages are self-consistent and can remain there indefinitely. However, any noise or other disturbance will cause  $A$  and  $B$  to switch to one of the two stable states. Figure 7.79(d) shows an analogy of a ball on a hill. The top of the hill is a metastable state. Any disturbance will cause the ball to roll down to one of the two stable states on the left or right side of the hill.

Figure 7.80(a) plots the output of the latch from Figure 7.17(g) as the data transitions near the falling clock edge. If the data changes at just the wrong time  $t_m$  within the aperture, the output can remain at the metastable point for some time before settling to a valid logic level. Figure 7.80(b) plots  $t_{DQ}$  vs.  $t_{DC} - t_m$  on a semilogarithmic scale for a rising input and output. The delay is less than or equal to  $t_{pdq}$  for inputs that meet the setup time and increases for inputs that arrive too close to  $t_m$ . The points marked on the graph will be used in the example at the end of this section.

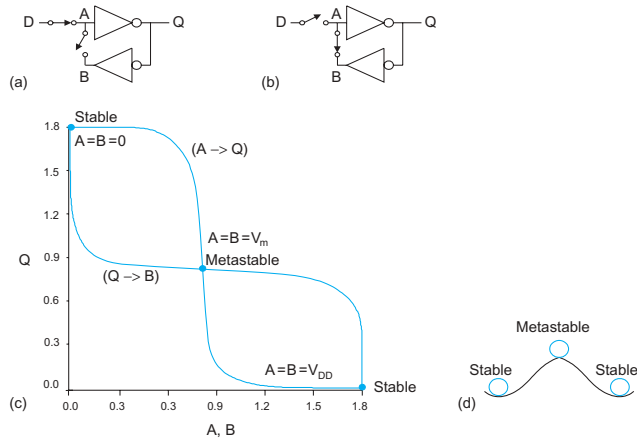


FIG 7.79 Metastable state in static latch

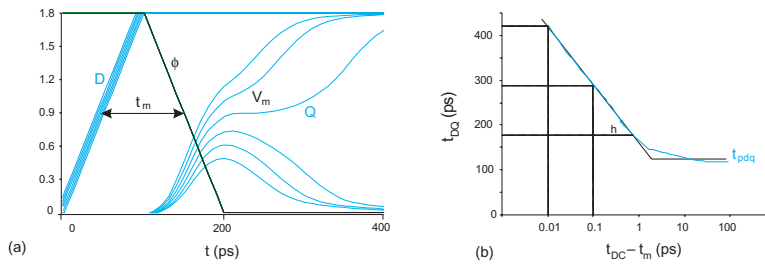
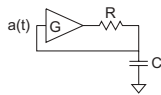


FIG 7.80 Metastable transients and propagation delay

The cross-coupled inverters behave like a linear amplifier with gain  $G$  when  $A$  is near the metastable voltage  $V_m$ . The inverter delay can be modeled with an output resistance  $R$  and load capacitance  $C$ . We can model the behavior in metastability by assuming that the initial voltage on node  $A$  when the latch becomes opaque at time  $t = 0$  is

$$A(0) = V_m + a(0) \tag{7.25}$$



**FIG 7.81** Small signal model of bistable element in metastability

where  $a(0)$  is a small signal offset from the metastable point. Figure 7.81 shows a small-signal model for  $a(t)$ . The behavior after time 0 is given by the first-order differential equation

$$\frac{Ga(t) - a(t)}{R} = C \frac{da(t)}{dt} \quad (7.26)$$

Solving this equation shows that the positive feedback drives  $a(t)$  exponentially away from the metastable point with a time constant determined by the gain and RC delay of the cross-coupled inverter loop.

$$a(t) = a(0)e^{\frac{t}{\tau_s}}; \tau_s = \frac{RC}{G-1} \quad (7.27)$$

Suppose the node is defined to reach a legal logic level when  $|a(t)|$  exceeds some deviation  $\Delta V$ . The time to reach this level is

$$t_{DQ} = \tau_s [\ln \Delta V - \ln a(0)] \quad (7.28)$$

This shows that the latch propagation delay increases as  $A(0)$  approaches the metastable point and  $a(0)$  approaches 0. The delay approaches infinity if  $a(0)$  is precisely 0, but this can never physically happen because of noise. However, there is no upper bound on the possible waiting time  $t$  required for the signal to become valid. If the input  $A(t)$  is a ramp that passes through  $V_m$  at time  $t_m$ ,  $a(0)$  is proportional to  $t_{DC} - t_m$ . Observe that EQ (7.28) is a good fit to the log-linear portion of Figure 7.80(b). The time constant  $\tau_s$  is essentially the reciprocal of the gain-bandwidth product [Flannagan85]. Therefore, the feedback loop in a latch should have a high gain-bandwidth product to resolve from metastability quickly.

Designers need to know the probability that latch propagation delay exceeds some time  $t'$ . Longer propagation delays are less likely because they require  $a(0)$  to be closer to 0. This probability should decrease with the clock period  $T_c$  because a uniformly distributed input change is less likely to occur near the critical time. Projecting through EQ (7.28) shows that it should also decrease exponentially with waiting time  $t'$ . Theoretical and experimental studies [Chaney83, Veendrick80, Horstmann89] find that the probability can be expressed as

$$P(t_{DQ} > t') = \frac{T_0}{T_c} e^{-\frac{t'}{\tau_s}} \text{ for } t' > b \quad (7.29)$$

where  $T_0$  and  $\tau_s$  can be extracted through simulation [Baghini02] or measurement. Intuitively,  $T_0/T_c$  describes the probability that the input would change during the aperture,

**Example**

Find  $\tau$ ,  $T_0$ , and  $b$  for the latch using the data in Figure 7.80.

**Solution:**  $b$  is the propagation delay above which the data fits a good straight line on a log-linear scale. In Figure 7.80, this appears to be approximately 175 ps. The probability that the delay exceeds some  $t'$  is the chance that the input changing at a random time falls within the small aperture that leads to the high delay. We can choose two points on the linear portion of the plot and solve for the two unknowns. For example, choosing (0.1 ps, 290 ps) and (0.01 ps, 415 ps), we solve

$$\begin{aligned}
 P(t_{DQ} > 290 \text{ ps}) &= \frac{0.1 \text{ ps}}{T_0} = \frac{T_0}{T_0} e^{-\frac{290 \text{ ps}}{\tau}} \\
 P(t_{DQ} > 415 \text{ ps}) &= \frac{0.01 \text{ ps}}{T_0} = \frac{T_0}{T_0} e^{-\frac{415 \text{ ps}}{\tau}}
 \end{aligned}
 \tag{7.30}$$

$T_0$  drops out of the equations and we find  $\tau = 54 \text{ ps}$  and  $T_0 = 21 \text{ ps}$ . Recall that this data was taken for a rising input. A conservative design should also consider the falling input and take data in the slow rather than typical environment.

causing metastability, and the exponential term describes the probability that the output hasn't resolved after  $t'$  if it did enter metastability. The model is only valid for sufficiently long propagation delays ( $b$  significantly greater than  $\Delta_{DQ}$ ).

We have seen that a good synchronizer latch should have a high-gain-bandwidth product. Conventional latches have data and clock transistors in series, increasing the delay (i.e., reducing the bandwidth). Figure 7.82 shows a synchronizer flip-flop in which the feedback loops simplify to cross-coupled inverter pairs [Dike99]. Furthermore, the flip-flop is reset to 0, and then is only set to 1 if  $D = 1$  to minimize loading on the feedback loop.

The flip-flop consists of master and slave jamb latches. Each latch is reset to 0 while  $D = 0$ . When  $D$  rises before  $\phi$ , the master output  $X$  is driven high. This in turn drives the slave output  $Q$  high when  $\phi$  rises. The pulldown transistors are just large enough to overpower the cross-coupled inverters, but should add as little stray capacitance to the feedback loops as possible.  $X$  and  $Q$  are buffered with small inverters so they do not load the feedback loops.

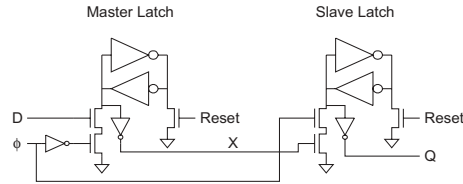


FIG 7.82 Fast synchronizer flip-flop

### 7.6.2 A Simple Synchronizer

A synchronizer accepts an input  $D$  and a clock  $\phi$ . It produces an output  $Q$  that ought to be valid some bounded delay after the clock. The synchronizer has an aperture defined by a setup and hold time around the rising edge of the clock. If the data is stable during the aperture,  $Q$  should equal  $D$ . If the data changes during the aperture,  $Q$  can be chosen arbitrarily. Unfortunately, it is impossible to build a perfect synchronizer because the duration of metastability can be unbounded. We define synchronizer failure as occurring if the output has not settled to a valid logic level after some time  $t'$ .

Figure 7.83 shows a simple synchronizer built from a pair of flip-flops.  $F1$  samples the asynchronous input  $D$ . The output  $X$  may be metastable for some time, but will settle to a good level with high probability if we wait long enough.  $F2$  samples  $X$  and produces an output  $Q$  that should be a valid logic level and be aligned with the clock. The synchronizer has a latency of one clock cycle,  $T_c$ . It can fail if  $X$  has not settled to a valid level by a setup time before the second clock edge.

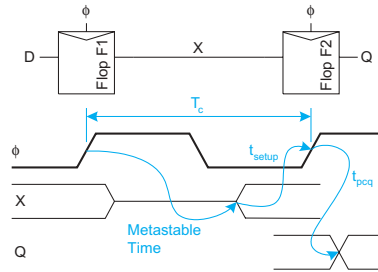


FIG 7.83 Simple synchronizer

Each flip-flop samples on the rising clock edge when the master latch becomes opaque. The slave latch merely passes along the contents of the master and does not sig-

**Example**

A particular synchronizer flip-flop in a 0.25 μm process has  $\tau_s = 20$  ps and  $T_0 = 15$  ps [Dike99]. Assuming the input toggles at  $N = 50$  MHz and the setup time is negligible, what is the minimum clock period  $T_c$  for which the MTBF exceeds one year?

**Solution:** 1 year  $\approx \pi \cdot 10^7$  seconds. Thus, we must solve

$$\pi \cdot 10^7 = \frac{T_c e^{\frac{T_c}{20 \cdot 10^{-12}}}}{(5 \cdot 10^7)(15 \cdot 10^{-12})} \quad (7.31)$$

numerically for a minimum clock period of 625 ps (1.6 GHz).

**Example**

How much longer must we wait for a 1000-year MTBF?

**Solution:** Solving a similar equation gives 760 ps. Increasing the waiting time by 135 ps improved MTBF by a factor of 1000.

nificantly affect the probability of metastability. If the synchronizer receives an average of  $N$  asynchronous input changes at  $D$  each second, the probability of synchronizer failure in any given second is

$$P(\text{failure}) = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau_s}} \quad (7.32)$$

and the mean time between failures increases exponentially with cycle time

$$MTBF = \frac{1}{P(\text{failure})} = \frac{T_c e^{\frac{T_c - t_{\text{setup}}}{\tau_s}}}{NT_0} \quad (7.33)$$

The acceptable MTBF depends on the application. For medical equipment where synchronizer reliability is crucial and latency is relatively unimportant, the MTBF can be chosen to be longer than the life of the universe ( $\sim 10^{19}$  seconds) by waiting more than one clock cycle before using the data. For noncritical applications, the MTBF can be chosen to be merely longer than the designer's expected duration of employment at the company!

### 7.6.3 Communicating Between Asynchronous Clock Domains

A common application of synchronizers is in communication between asynchronous clock domains, i.e., blocks of circuits that do not share a common clock. Suppose System A is controlled by  $clkA$  that needs to transmit  $N$ -bit data words to System B, which is controlled by  $clkB$ , as shown in Figure 7.84. The systems can represent separate chips or separate units within a chip using unrelated clocks. Each word should be received by system B exactly once. System A must guarantee that the data is stable while the flip-flops in System B sample the word. It indicates when new data is valid by using a request signal ( $Req$ ), so System B receives the word exactly once rather than zero or multiple times. System B replies with an acknowledge signal ( $Ack$ ) when it has sampled the data so System A knows when the data can safely be changed. If the relationship between  $clkA$  and  $clkB$  is completely unknown, a synchronizer is required at the interface.

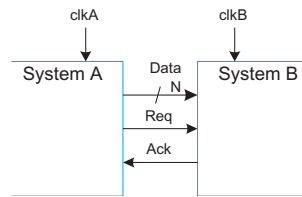
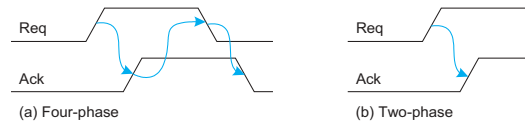


FIG 7.84 Communication between asynchronous systems

The request and acknowledge signals are called *handshaking* lines. Figure 7.85 illustrates two-phase and four-phase handshaking protocols. The four-phase handshake is level-sensitive while the two-phase handshake is edge-triggered. In the four-phase handshake, system A places data on the bus. It then raises  $Req$  to indicate that the data is valid. System B samples the data when it sees a high value on  $Req$  and raises  $Ack$  to indicate that the data has been captured. System A lowers  $Req$ , then system B lowers  $Ack$ . This protocol requires four transitions of the handshake lines. In the two-phase handshake, system A places data on the bus. Then it changes  $Req$  (low to high or high to low) to indicate that the data is valid. System B samples the data when it detects a change in the level of  $Req$  and toggles  $Ack$  to indicate that the data has been captured. This protocol uses fewer transitions (and thus possibly less time and energy), but requires circuitry that responds to edges rather than levels.



**FIG 7.85** Four-phase and two-phase handshake protocols

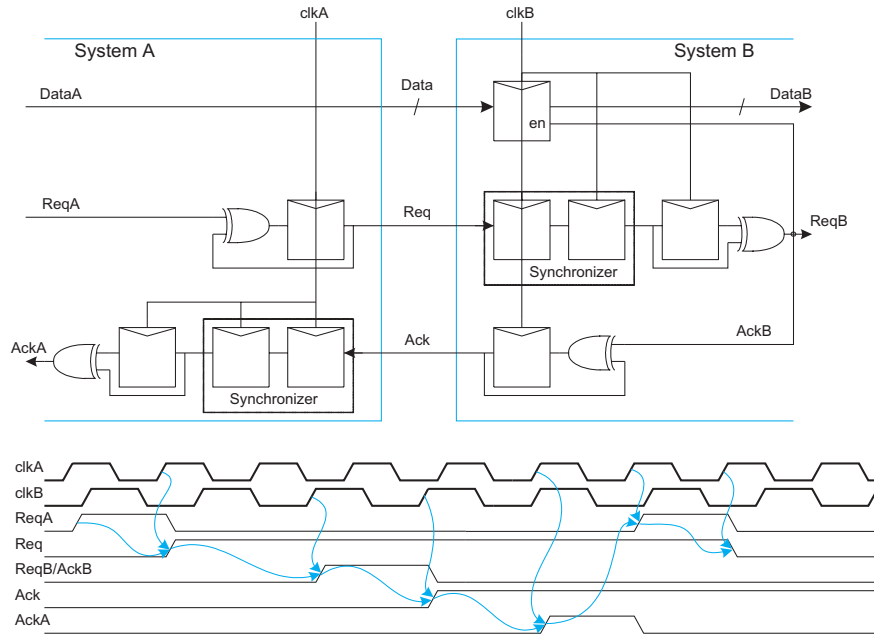
*Req* is not synchronized to *clkB*. If it changes at the same time *clkB* rises, System *B* may receive a metastable value. Thus, System *B* needs a synchronizer on the *Req* input. If the synchronizer waits long enough, the request will resolve to a valid logic level with very high probability. The synchronizer may resolve high or low. If it resolves high, the rising request was detected and System *B* can sample the data. If it resolves low, the rising request was just missed. However, it will be detected on the next cycle of *clkB*, just as it would have been if the rising request occurred just slightly later. *Ack* is not synchronized to *clkA*, so it also requires a synchronizer.

Figure 7.86 shows a typical two-phase handshaking system [Crews03]. *clkA* and *clkB* operate at unrelated frequencies and each system may not know the frequency of its counterpart. Each system contains a synchronizer, a level-to-pulse converter, and a pulse-to-level converter. System *A* asserts *ReqA* for one cycle when *DataA* is ready. We will refer to this as a *pulse*. The XOR and flip-flop form a pulse-to-level converter that toggles the level of *Req*. This level is synchronized to *clkB*. When an edge is detected, the level-to-pulse converter produces a pulse on *ReqB*. This pulse in turn toggles *Ack*. The acknowledge level is synchronized to *clkA* and converted back to a pulse on *AckA*. The synchronizers add significant latency so the throughput of asynchronous communication can be much lower than that of synchronous communication.

#### 7.6.4 Common Synchronizer Mistakes

Although a synchronizer is a simple circuit, it is notoriously easy to misuse. For example, the AMD 9513 system timing controller, AMD 9519 interrupt controller, Zilog Z-80 Serial I/O interface, Intel 8048 microprocessor, and AMD 29000 microprocessor are all said to have suffered from metastability problems [Wakerly00].

One way to build a bad synchronizer is to use a bad latch or flip-flop. The synchronizer depends on positive feedback to drive the output to a good logic level. Therefore, dynamic latches without feedback such as Figure 7.17(a–d) do not work. The probability of failure grows exponentially with the time constant of the feedback loop. Therefore, the loop should be lightly loaded. The latch from Figure 7.17(f) is a poor choice because a large capacitive load on the output will increase the time constant; Figure 7.17(g) is a much better choice.



**FIG 7.86** Two-phase handshake circuitry with synchronizers

Another error is to capture inconsistent data. For example, Figure 7.87(a) shows a single signal driving two synchronizers (each consisting of a pair of back-to-back flip-flops). If the signal is stable through the aperture,  $Q1$  and  $Q2$  will be the same. However, if the signal changes during the aperture,  $Q1$  and  $Q2$  might resolve to different values. If the system requires that  $Q1$  and  $Q2$  be identical representations of the data input, they must come from a single synchronizer.

Another example is to synchronize a multi-bit word where more than one bit might be changing at a time. For example, if the word in Figure 7.87(b) is transitioning from 0000 to 1111, the synchronizer might produce a value such as 0101 that is neither the old nor the new data word. For this reason, the system in Figure 7.86 synchronized only the *Req/Ack* signals and used them to indicate that data was stable to sample or finished being sampled. *Gray codes* (see Section 10.7.3) are also useful for counters whose outputs must be synchronized because exactly one bit changes on each count so the synchronizer is guaranteed to find either the old or the new data value.

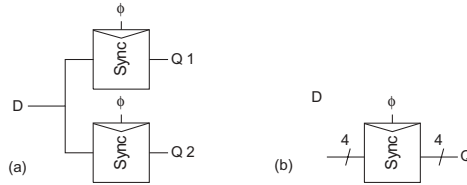


FIG 7.87 Bad synchronizer designs

In general, synchronizer bugs are intermittent and very difficult to locate and diagnose. For this reason, the number of synchronizers in a system should be strictly limited.

### 7.6.5 Arbiters

The arbiter of Figure 7.88(a) is closely related to the synchronizer. It determines which of two inputs arrived first. If the spacing between the inputs exceeds some aperture time, the first input should be acknowledged. If the spacing is smaller, exactly one of the two inputs should be acknowledged, but the choice is arbitrary. For example, in a television game show, two contestants may pound buttons to answer a question. If one presses the button first, she should be acknowledged. If both press the button at times too close to distinguish, the host may choose one of the two contestants arbitrarily.

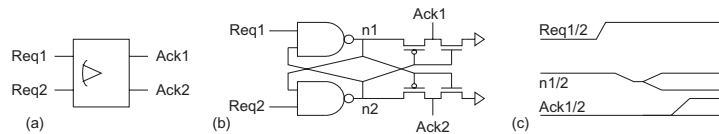


FIG 7.88 Arbiter

Figure 7.88(b) shows an arbiter built from a SR latch and a four-transistor metastability filter. If one of the request inputs arrives well before the other, the latch will respond appropriately. However, if they arrive at nearly the same time, the latch may be driven into metastability, as shown in Figure 7.88(c). The filter keeps both acknowledge signals low until the voltage difference between the internal nodes  $n1$  and  $n2$  exceeds  $V_t$ , indicating that a decision has been made. Such an asynchronous arbiter will never produce metastable outputs. However, the time required to make the decision can be unbounded, so the acknowledge signals must be synchronized before they are used in a clocked system.

Arbiters can be generalized to select 1-of- $N$  or  $M$ -of- $N$  inputs. However, such arbiters have multiple metastable states and require careful design [van Berkel99].



### 7.6.6 Degrees of Synchrony

The simple synchronizer from Section 7.6.2 accepts inputs that can change at any time, but has a nonzero probability of failure. In practice, many inputs may not be aligned to a single system clock, but they may still be predictable. Table 7.3 provides a classification of degrees of synchrony between input signals and the receiver system clock [Messerschmitt90] based on the difference in phase  $\Delta\phi$  and frequency  $\Delta f$ .

[Dally98] describes a number of synchronizers that have zero failure probability and possibly lower latency when the input is predictable. They are based on the observation that either the signal or a copy of the signal delayed by  $t_d$  will be stable throughout the aperture. Hence, a synchronizer that can predict the input arrival time can choose the signal or its delayed counterpart to safely sample. Mesochronous signals are synchronized by measuring the phase difference and delaying the input enough to ensure it falls outside the aperture. Plesiochronous signals can be synchronized in a similar fashion, but the phase difference slowly varies so the delay must be occasionally adjusted. Because the frequencies differ, the synchronizer requires some control flow to handle the missing or extra data items. Periodic signals also require control flow and use a clock predictor to calculate where the next clock edge will occur and whether the signal must be delayed to avoid falling in the aperture.



## 7.7 Wave Pipelining

Recall that sequencing elements are used in pipelined systems to prevent the current token from overtaking the next token or from being overtaken by the previous token in the pipeline. If the elements propagate through the pipeline at a fairly constant rate, explicit sequencing elements may not be necessary to maintain sequence. As an analogy, fiber optic cables carry data as a series of light pulses. Many pulses enter the cable before the first one reaches the end, yet the cable does not need internal latches to keep the pulses separated because they propagate along the cable at a well-controlled velocity. The maximum data rate is limited by the dispersion along the line that causes pulses to smear over time and blur into one another if they become too short.

Figure 7.89 compares traditional pipelining with wave pipelining. In both cases, the pipeline contains combinational logic separated by registers (Figure 7.89(a)). The registers  $F1$  and  $F2$  receive clocks  $clk1$  and  $clk2$  that are nominally identical, but might experience skew. Figure 7.89(b) shows traditional pipelining. The data is launched on the rising edge of  $clk1$ . Its propagation is indicated by the hashed cone.  $D2$  becomes stable somewhere between the contamination and propagation delays after the clock edge (neglecting the flip-flop  $clk$ -to- $Q$  delay).  $D2$  must not change during the setup and hold aperture around  $clk2$ , marked with the gray box. The figure shows two successive cycles in which tokens  $i$

Table 7.3 Degrees of synchrony				
Classification	Periodic	$\Delta\phi$	$\Delta f$	Description
<b>Synchronous</b>	Yes	0	0	Signal has same frequency and phase as clock. Safe to sample signal directly with the clock. <b>Example:</b> Flip-flop to flip-flop on chip.
<b>Mesochronous</b>	Yes	constant	0	Signal has same frequency, but is out of phase with the clock. Safe to sample signal if it is delayed by a constant amount to fall outside aperture. <b>Example:</b> Chip-to-chip where chips use same clock signal, but might have arbitrarily large skews.
<b>Plesiochronous</b>	Yes	varies slowly	small	Signal has nearly the same frequency. Phase drifts slowly over time. Safe to sample signal if it is delayed by a variable but predictable amount. Difference in frequency can lead to dropped or duplicated data. <b>Example:</b> Board-to-board where boards use clock crystals with small mismatches in nominally identical rates.
<b>Periodic</b>	Yes	varies rapidly	large	Signal is periodic at an arbitrary frequency. Periodic nature can be exploited to predict and delay accordingly when data will change during aperture. <b>Example:</b> Board-to-board where boards use different frequency clocks.
<b>Asynchronous</b>	No	unknown	unknown	Signal may change at arbitrary times. Full synchronizer is required. <b>Example:</b> Input from pushbutton switch.

and  $i + 1$  move through the pipeline. Each token passes through the combinational logic in a single cycle. Figure 7.89(c) shows wave pipelining with a clock of twice the frequency. Token  $i$  enters the combinational logic, but takes two cycles to reach  $F2$ . Meanwhile, token  $i + 1$  enters the logic a cycle later. As long as each token is stable to sample at  $F2$  and the cones do not overlap, the pipeline will operate correctly with the same latency but twice the throughput.

[Burleson98] gives a tutorial on wave pipelining and derives the timing constraints. In general, a wave pipeline can contain  $N$  tokens between each pair of registers. The maximum value of  $N$  is limited by the ratio of propagation delay to dispersion of the logic cones

$$N < \frac{t_{pd}}{t_{pd} - t_{cd}} \quad (7.34)$$

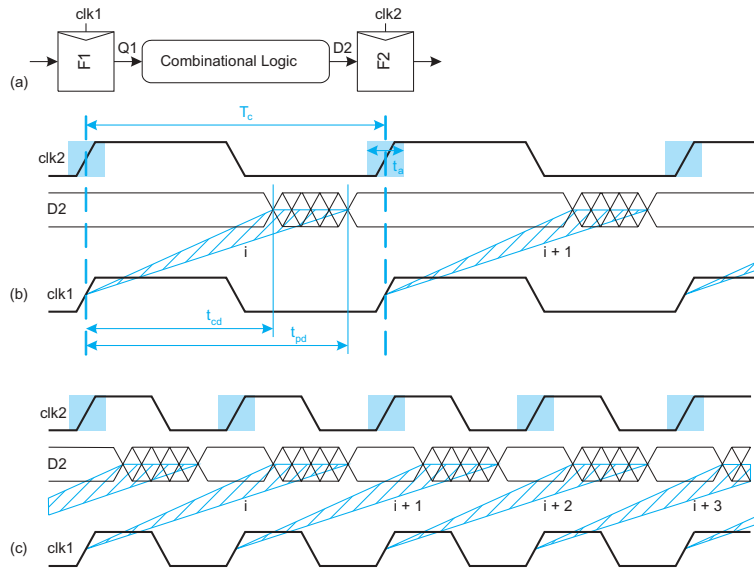


FIG 7.89 Wave pipelining

If the contamination and propagation delays are nearly equal, the combinational logic can contain many tokens simultaneously. In practice, the delays tend to be widely variable because of voltage, temperature, and processing as well as differences in path lengths through the logic. Clock skew and sequencing overhead also eat into the timing budgets. In practice, even achieving  $N = 2$  simultaneous tokens can be difficult and wave pipelining has not achieved widespread popularity for general-purpose logic.

## 7.8 Pitfalls and Fallacies

### Incompletely reporting flip-flop delay

The effective delay of a flip-flop is its minimum D-to-Q time. This is the sum of the setup time  $t_{\text{setup}}$  and the clk-to-Q delay  $t_{\text{pdq}}$  if these delays are defined to minimize the sum. Some engineers focus on only the clk-to-Q delay or define setup and clk-to-Q delays in a way that does not minimize the sum.

### Failing to check hold times

One of the leading reasons that chips fail to operate even though they appear to simulate correctly is hold time violations, especially violations caused by unexpected clock skew. Unless a design uses two-phase non-overlapping clocks, the clock skew should be carefully modeled and the hold times should be checked with a static timing analyzer. These checks should happen as soon as a block is designed so that errors can be corrected immediately. For example, a large microprocessor used a wide assortment of delayed clocks to solve setup time problems on long paths. Hold times were not checked until shortly before tapeout, leading to a significant schedule slip when many violations were found.

### Choosing a sequencing methodology too late in the design cycle

Designers may choose from many sequencing methodologies, each of which has tradeoffs. The best methodology for a particular application is very debatable, and engineers love a good debate. If the sequencing methodology is not settled at the beginning of the project, experience shows that engineers will waste tremendous amounts of time redoing work as the method changes, or supporting and verifying multiple methodologies. Projects need a strong technical manager to demand that a team choose one method at the beginning and stick with it.

### Failing to synchronize asynchronous inputs

Unsynchronized inputs can cause strange and wonderful sporadic system failures that are very difficult to locate. For example, a finite state machine running off one clock received a *READY* input from a *UART* running on another clock when the *UART* had data available, as shown in Figure 7.90. The designer reasoned that synchronizing the *READY* signal was unimportant because if it changed near the clock edge of the FSM, she did not care whether it was detected in one cycle or the next. Moreover, the clock was so slow that metastability would have time to resolve. However, the FSM occasionally failed by jumping to seemingly random states that could never legally occur. After two months of debugging, she realized that the problem was triggered if the asynchronous *READY* signal was asserted a few gate delays before the FSM clock edge. The propagation delay through the combinational logic was different for various bits of the next state logic. Some bits had changed to their new values while others were still at their old values, so the FSM could jump to an undefined state. Registering the *READY* signal with the FSM clock before it drove the combinational logic solved the problem.

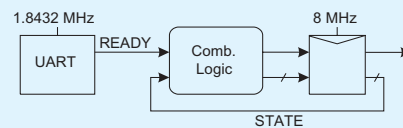


FIG 7.90 Unsynchronized input

### Building faulty synchronizers

Designers have found many ways to build faulty synchronizers. For example, if an asynchronous input drives more than one synchronizer, the two synchroniz-

*continued*

ers can resolve to different values. If they must produce consistent outputs, only one synchronizer should be used. In another example, synchronizers must not accept multi-bit inputs where more than one of the bits can change simultaneously. This would pose the risk that some of the bits resolve as changed while others re-

solve in their old state, resulting in an invalid pattern that is neither the old nor the new input word. In yet another example, synchronizers with poorly designed feedback loops can be much slower than expected and can have exponentially worse mean time between failures.

## 7.9 Case Study: Pentium 4 and Itanium 2 Sequencing Methodologies

The Pentium 4 and Itanium 2 represent two philosophies of high-performance microprocessor design sometimes called *Speed Demon* and *Braniac*, respectively. The Pentium 4 was designed by Intel for server and desktop applications and has migrated into laptop computers as well. The Itanium 2 was jointly designed by Hewlett-Packard and Intel for high-end server applications. Figure 7.91 shows the date of introduction and the performance of several generations of these processors.

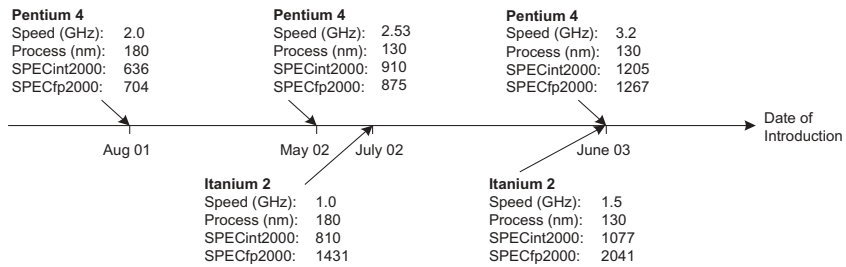


FIG 7.91 Microprocessor timeline

The Pentium 4 uses a very long (20+ stage) pipeline with few stages of logic per cycle to achieve extremely high frequencies. It issues up to three instructions per cycle, but the long pipeline causes severe penalties for branch mispredictions and cache misses, so the overall average number of instructions executed per cycle is relatively low. Figure 4.76 showed a die photo of the 42-million transistor Pentium 4. The chip consumes around 55 watts. A top-of-the-line Pentium 4 sold in 1000-unit quantities for around \$400–\$600 (depending on price pressure from competitor AMD). The chip has aggressively migrated

into Intel's most advanced processes both to achieve high performance and to reduce the die size and manufacturing cost. The Speed Demon approach also gives Intel bragging rights to the highest clock frequency microprocessors, which is important because many consumers compare processors on clock frequency rather than benchmark performance.

In contrast, the Itanium 2 focuses on executing many instructions per cycle at a lower clock rate. It uses an 8-stage integer pipeline clocked at about half the rate of the Pentium 4 in the same process, so each cycle accommodates about twice as many gate delays (roughly 20–24 FO4 inverter delays, compared to roughly 10–12 for the Pentium 4). However, it issues up to six instructions per cycle and has a very high-bandwidth memory and I/O system to deliver these instructions and their data. As a result, it achieves nearly the same integer performance and much better floating-point benchmark results than the Pentium 4. Moreover, it also performs well on multiprocessor and transaction processing tasks typical of high-end servers. Figure 7.92 shows a die photo of the Itanium 2 with a 3MB level 3 (L3) cache; notice that the three levels of cache occupy most of the die area and most of the 221 million transistors. The 1.5 GHz model with 6MB cache bumps the transistor count to 410 million and further dwarfs the processor core. The chip consumes about 130 watts, limited by the cost of cooling multiprocessor server boxes. A high-end Itanium 2 sold for more than \$4000 because the server market is much less price-sensitive. The chip has lagged a year behind the Pentium 4 in process technology.

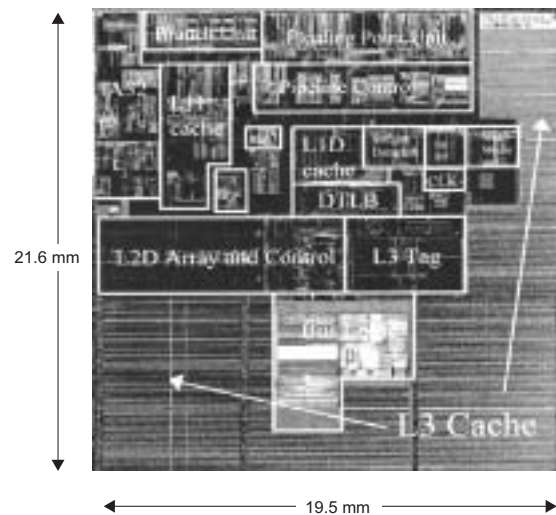


FIG 7.92 Itanium II die photo

### 7.9.1 Pentium 4 Sequencing

The Pentium 4 actually operates at three different internal clock rates [Hinton01, Kurd01]. In addition to the *core clock* that drives most of the logic, it has a double-speed *fast clock* for the ALU core and a half-speed *slow clock* for noncritical portions of the chip. The core clock is distributed across the chip using a triple spine, as will be shown in Section 12.5.4.3. These clocks drive pulsed latches, flip-flops, and self-resetting domino gates.

The ALU runs at a remarkable rate of twice the core clock frequency (about 6 FO4 inverter delays). To achieve this speed, it is stripped down to just the essential functions of the bypass multiplexer and the 16-bit add/subtract unit. Other less commonly used blocks such as the shifter and multiplier operate at core frequency. The ALU uses unfooted domino gates. The gates produce pulsed outputs and precharge in a self-timed fashion using the Global STP approach described in Section 7.5.2.4. These circuits demanded extensive verification by expert circuit designers to ensure the domino gates function reliably.

The Pentium 4 uses pulsed latches operating at all three clock speeds. Figure 7.93 shows pulse generators that receive the core clock and produce the appropriate output pulses. The medium-speed pulse generator produces a pulse on the rising edge of the core clock. The pulse width can be shaped by the adjustable delay buffer to provide both long pulses (offering more time borrowing) and short pulses (to prevent hold-time problems). The buffer is built from a digitally controlled current-starved inverter with four discrete settings. The pulse generator also accepts enable signals to gate the clock or save power on unused blocks. The slow pulse generator produces a pulse on every other rising edge of the core clock. To do this, it receives a sync signal that is asserted every other cycle. While the sync signal must be distributed globally, it is more convenient than distributing a half-speed clock because it can accept substantial skew while still being stable around the clock edge. The fast pulse generator produces pulses on both the rising and falling edges of the core clock. Therefore, the core clock should have nearly equal high and low times, i.e., 50% duty cycle, so the pulses are equally spaced.

### 7.9.2 Itanium 2 Sequencing

The Itanium 2 operates at a single primary clock speed, but also makes use of extensive domino logic and pulsed latches [Naffziger02, Fetzer02, Rusu03]. The clock is distributed across the chip using an H-tree, as will be shown in Section 12.5.4.2. The H-tree drives 33 second-level clock buffers distributed across the chip. These buffer outputs, called SLCBOs, in turn drive local clock gaters that serve banks of sequencing elements within functional blocks. There are 24 different types of clock gaters producing inverted, stretched, delayed, and pulsed clocks. Figure 7.94 shows some of these clocks. Each gater comes in many sizes and is tuned to drive different clock loads with low skew over regions of up to about 1000  $\mu\text{m}$ . Section 12.5.6.3 analyzes the clock skew from this distribution network.

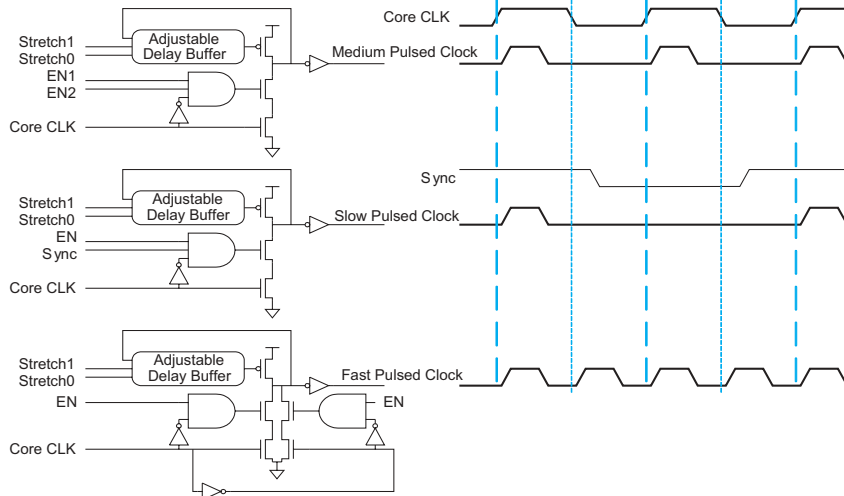


FIG 7.93 Pulse generators

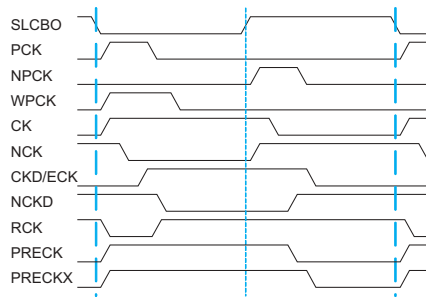


FIG 7.94 Clock gater waveforms



## Summary

This chapter has examined the tradeoffs of sequencing with flip-flops, two-phase transparent latches, and pulsed latches. The ITRS forecasts cycle times dropping well below 10 FO4 delays (see Table 4.17). Minimizing sequencing overhead will be very important in these high-performance systems. Flip-flops are the simplest, but have the greatest sequencing overhead. Transparent latches are most tolerant of skew and allow the most time borrowing, but require greater design effort to partition logic into half-cycles instead of cycles. Pulsed latches have the lowest sequencing overhead, but are most susceptible to min-delay problems. Table 7.4 compares the sequencing overhead, minimum delay constraint, and time borrowing capability of each technique. All of the techniques are used in commercial products, and the designer's choice depends on the design constraints and CAD tools.

**Table 7.4** Comparison of sequencing elements

	Sequencing overhead ( $T_c - t_{pd}$ )	Minimum logic delay $t_{cd}$	Time borrowing $t_{borrow}$
Flip-Flops	$t_{pcq} + t_{setup} + t_{skew}$	$t_{hold} - t_{cq} + t_{skew}$	0
Two-Phase Transparent Latches	$2t_{pdq}$	$t_{hold} - t_{cq} - t_{nonoverlap} + t_{skew}$ in each half-cycle	$\frac{T_c}{2} - (t_{setup} + t_{nonoverlap} + t_{skew})$
Pulsed Latches	$\max(t_{pdq}, t_{pcq} + t_{setup} - t_{pw} + t_{skew})$	$t_{hold} - t_{cq} + t_{pw} + t_{skew}$	$t_{pw} - (t_{setup} + t_{skew})$

In class projects for introductory VLSI classes, timing analysis is often rudimentary or nonexistent. Using two-phase nonoverlapping clocks generated off chip is attractive because you can guarantee the chip will have no max-delay or min-delay failures if the clock period and nonoverlap are sufficiently large. However, it is not practical to generate and distribute two nonoverlapping phases on a large, high-performance commercial chip.

The great majority of low- and mid-performance designs and some high-speed designs use flip-flops. Flip-flops are very easy to use and are well understood by most designers. Even more importantly, they are handled well by synthesis tools and timing analyzers. Unfortunately, in systems with few gate delays per cycle, the sequencing overhead can consume a large fraction of the cycle. Moreover, many standard cell flip-flops are intentionally rather slow to prevent hold time violations at the expense of greater sequencing overhead.

Most two-phase latch systems distribute a single clock and locally invert it to drive the second latch. These systems tolerate significant amounts of clock skew without loss of performance and can borrow time to balance delay intentionally or opportunistically. However, the systems require more effort to understand because time borrowing distrib-

utes the timing constraints across many stages of a pipeline rather than isolating them at each stage. Not all timing analyzers handle latches gracefully, especially when there are different amounts of clock skew between different clocks [Harris99]. Two-phase latches have been used in the Alpha 21064 and 21164 [Gronowski98], PowerPC 603 [Gerosa94], and many other IBM designs.

Pulsed latches have low sequencing overhead. They present a tradeoff when choosing pulse width: A wide pulse permits more time borrowing and skew tolerance, but makes min-delay constraints harder to meet. Pulsed latches are also popular because they can be modeled as fast flip-flops with a lousy hold time from the point of view of a timing analyzer (or novice designer) if intentional time borrowing is not permitted. The min-delay problems can be largely overcome by mixing pulsed latches for long paths and flip-flops for short paths. Unfortunately, many real designs have paths in which the propagation delay is very long but the contamination delay is very short, making robust design more challenging. Pulsed latches have been used on Itanium 2 [Naffziger03], Athlon [Draper97], and CRAY 1 [Unger86]. However, they can wreak havoc with conventional commercially available design flows and are best avoided unless the performance requirements are extreme.

Domino circuits are widely used in high-performance systems because they are 1.5-2x faster than static CMOS. Traditional domino circuits with latches have high sequencing overhead that wastes much of the potential speedup, so most designers have moved to skew-tolerant techniques. Static-to-domino interfaces impose hard edges and the associated sequencing overhead, motivating the use of domino throughout critical loops. Single-rail domino only computes noninverting functions, so most loops require dual-rail domino that consumes more area, wiring, and power and is ill-suited to wide NORs. An alternative is to push the inverting functions to the end of the pipeline, using single-rail domino through most of the pipeline and nonmonotonic static logic at the end. The area savings comes at the cost of one hard edge in the cycle.

Four-phase or delayed reset skew-tolerant domino circuits work well in datapaths because the clock generation is relatively simple. Self-resetting domino is ideally suited to memories where the decoder power consumption is greatly reduced by only precharging the output that switched and where the number of unique circuits to design is relatively small. It was also used on the Pentium 4, but was costly in terms of designer effort because so many pulse constraints must be satisfied.

Clock-delayed domino is used in wide dynamic NOR functions where the power consumption of pseudo-nMOS is unacceptable. For example, it is an important technique for CAMs and PLAs. The delay matching raises an unpleasant tradeoff between speed and correct operation, requiring significant margin for safe operation. The risk of race conditions deters many designers from using it more widely. Annihilation gates and complementary signal generators are interesting special cases in which no clock gate delay at all is required. Output prediction logic is also interesting, but has yet to be proven in a large application.

When inputs to a system arrive asynchronously, they cannot be guaranteed to meet setup or hold times at clocked elements. Even if we do not care whether an input arrived in one cycle or the next, we must ensure that the clocked element produces a valid logic

level. Unfortunately, if the element samples a changing input at just the wrong time, it may produce a metastable output that remains invalid for an unbounded amount of time. The probability of metastability drops off exponentially with time. Systems use synchronizers to sample the asynchronous input and hold it long enough to resolve to a valid logic level with very high probability before passing it onward.

Most synchronous VLSI systems uses opaque sequencing elements to separate one token from the next. In contrast, many optical systems transmit data as pulses separated in time. As long as the propagation medium does not disperse the pulses too badly, they can be recovered at a receiver. Similarly, if a VLSI system has low dispersion, i.e., nearly equal contamination and propagation delays, it can send more than one wave of data without explicit latching. Such wave pipelining offers the potential of high throughput and low sequencing overhead. However, it is difficult to perform in practice because of the variability of data delay.

## Exercises

Use the following timing parameters for the questions in this section.

**Table 7.5** Sequencing element parameters

	Setup Time	<i>clk</i> -to- <i>Q</i> Delay	<i>D</i> -to- <i>Q</i> Delay	Contamination Delay	Hold Time
Flip-flops	65 ps	50 ps	n/a	35 ps	30 ps
Latches	25 ps	50 ps	40 ps	35 ps	30 ps

- 7.1 For each of the following sequencing styles, determine the maximum logic propagation delay available within a 500 ps clock cycle. Assume there is zero clock skew and no time borrowing takes place.
  - a) Flip-flops
  - b) Two-phase transparent latches
  - c) Pulsed latches with 80 ps pulse width
- 7.2 Repeat Exercise 7.1 if the clock skew between any two elements can be up to 50 ps.
- 7.3 For each of the following sequencing styles, determine the minimum logic contamination delay in each clock cycle (or half-cycle, for two-phase latches). Assume there is zero clock skew.
  - a) Flip-flops
  - b) Two-phase transparent latches with 50% duty cycle clocks

- c) Two-phase transparent latches with 60 ps of nonoverlap between phases
  - d) Pulsed latches with 80 ps pulse width
- 7.4 Repeat Exercise 7.3 if the clock skew between any two elements can be up to 50 ps.
- 7.5 Suppose one cycle of logic is particularly critical and the next cycle is nearly empty. Determine the maximum amount of time the first cycle can borrow into the second for each of the following sequencing styles. Assume there is zero clock skew.
- a) Flip-flops
  - b) Two-phase transparent latches with 50% duty cycle clocks
  - c) Two-phase transparent latches with 60 ps of nonoverlap between phases
  - d) Pulsed latches with 80 ps pulse width
- 7.6 Repeat Exercise 7.5 if the clock skew between any two elements can be up to 50 ps.
- 7.7 Prove EQ (7.17).
- 7.8 Consider a flip-flop built from a pair of transparent latches using nonoverlapping clocks. Express the setup time, hold time, and clock-to- $Q$  delay of the flip-flop in terms of the latch timing parameters and  $t_{\text{nonoverlap}}$ .
- 7.9 For the path in Figure 7.97, determine which latches borrow time and if any setup time violations occur. Repeat for cycle times of 1200, 1000, and 800 ps. Assume there is zero clock skew and that the latch delays are accounted for in the propagation delay  $\Delta$ 's.
- a)  $\Delta_1 = 550$  ps;  $\Delta_2 = 580$  ps;  $\Delta_3 = 450$  ps;  $\Delta_4 = 200$  ps
  - b)  $\Delta_1 = 300$  ps;  $\Delta_2 = 600$  ps;  $\Delta_3 = 400$  ps;  $\Delta_4 = 550$  ps

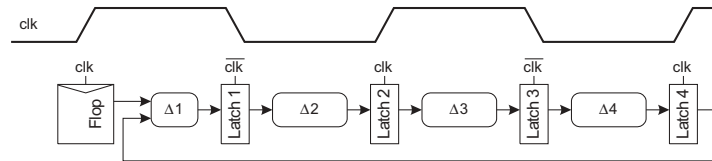


FIG 7.97 Example path

- 7.10 Determine the minimum clock period at which the circuit in Figure 7.98 will operate correctly for each of the following logic delays. Assume there is zero clock skew and that the latch delays are accounted for in the propagation delay  $\Delta$ 's.
- a)  $\Delta_1 = 300$  ps;  $\Delta_2 = 400$  ps;  $\Delta_3 = 200$  ps;  $\Delta_4 = 350$  ps

- b)  $\Delta_1 = 300$  ps;  $\Delta_2 = 400$  ps;  $\Delta_3 = 400$  ps;  $\Delta_4 = 550$  ps
- c)  $\Delta_1 = 300$  ps;  $\Delta_2 = 900$  ps;  $\Delta_3 = 200$  ps;  $\Delta_4 = 350$  ps

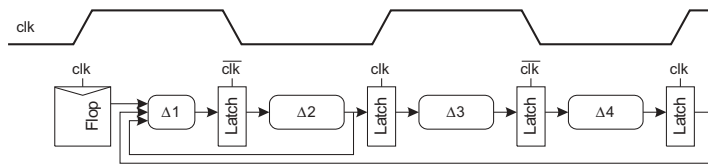


FIG 7.98 Another example path

- 7.11 Repeat Exercise 7.10 if the clock skew is 100 ps.
- 7.12 Label the timing types of each signal in the circuit from Figure 7.97. The flip-flop is constructed with back-to-back transparent latches—the first controlled by  $clk\_b$  and the second by  $clk$ .
- 7.13 Using a simulator, compare the  $D$ -to- $Q$  propagation delays of a conventional dynamic latch from Figure 7.17(d) and a TSPC latch from Figure 7.30(a). Assume each latch is loaded with a fanout of 4. Use  $4\lambda$ -wide clocked transistors and tune the other transistor sizes for least propagation delay.
- 7.14 Using a simulator, find the setup and hold times of a TSPC latch under the assumptions of Exercise 7.13.
- 7.15 Determine the maximum logic propagation delay available in a cycle for a traditional domino pipeline using a 500 ps clock cycle. Assume there is zero clock skew.
- 7.16 Repeat Exercise 7.15 if the clock skew between any two elements can be up to 50 ps.
- 7.17 Determine the maximum logic propagation delay available in a cycle for a four-phase skew-tolerant domino pipeline using a 500 ps clock cycle. Assume there is zero clock skew.
- 7.18 Repeat Exercise 7.17 if the clock skew between any two elements can be up to 50 ps.
- 7.19 How much time can one phase borrow into the next in Exercise 7.18 if the clocks each have a 50% duty cycle?
- 7.20 Repeat Exercise 7.18 if the clocks have a 65% duty cycle.
- 7.21 Design a fast-pulsed latch. Make the gate capacitance on the clock and data inputs equal. Let the latch drive an output load of four identical latches. Simulate your latch and find the setup and hold times and clock-to- $Q$  propagation and contamination delays. Express your results in FO4 inverter delays.

- 7.22 Simulate the worst-case propagation delay of an 8-input dynamic NOR gate driving a fanout of 4. Report the delay in all 16 design corners (voltage, temperature, nMOS, pMOS). Also determine the delay of a fanout-of-4 inverter in each of these corners. By what percentage does the absolute propagation delay of the NOR gate vary across corners? By what percentage does its normalized delay vary (in terms of FO4 inverters)? Comment on the implications for circuits using matched delays.
- 7.23 A synchronizer uses a flip-flop with  $\tau_i = 54$  ps and  $T_0 = 21$  ps. Assuming the input toggles at 10 MHz and the setup time is negligible, what is the minimum clock period for which the mean time between failures exceeds 100 years?
- 7.24 Simulate the synchronizer flip-flop of Figure 7.82 and make a plot analogous to Figure 7.80. From your plot, find  $\Delta_{DQ}$ ,  $b$ ,  $\tau$ , and  $T_0$ .
- 7.25 InferiorCircuits, Inc., wants to sell you a perfect synchronizer that they claim never produces a metastable output. The synchronizer consists of a regular flip-flop followed by a high-gain comparator that produces a high output for inputs above  $0.25 \cdot V_{DD}$  and a low output for inputs below that point. The VP of marketing argues that even if the flip-flop enters metastability, its output will hover near  $V_{DD}/2$  so the synchronizer will produce a good high output after the comparator. Why wouldn't you buy this synchronizer?