Impact of clock skew and jitter on performance and functionality

Alternative timing methodologies

Synchronization issues in digital IC and board design

Clock generation

10.1 Introduction
10.2 Classification of Digital Systems
10.3 Synchronous Design — An In-depth Perspective
10.3.1 Synchronous Timing Basics
10.3.2 Sources of Skew and Jitter
10.3.3 Clock-Distribution Techniques
10.3.4 Latch-Based Clocking
10.4 Self-Timed Circuit Design*
10.5 Synchronizers and Arbiters*
10.6 Clock Synthesis and Synchronization Using a Phase-Locked Loop
10.7 Future Directions
10.8 Perspective: Synchronous versus Asynchronous Design
10.9 Summary
10.10 To Probe Further
10.1 Introduction

All sequential circuits have one property in common—a well-defined ordering of the switching events must be imposed if the circuit is to operate correctly. If this were not the case, wrong data might be written into the memory elements, resulting in a functional failure. The synchronous system approach, in which all memory elements in the system are simultaneously updated using a globally distributed periodic synchronization signal (that is, a global clock signal), represents an effective and popular way to enforce this ordering. Functionality is ensured by imposing some strict constraints on the generation of the clock signals and their distribution to the memory elements distributed over the chip; non-compliance often leads to malfunction.

This Chapter starts with an overview of the different timing methodologies. The majority of the text is devoted to the popular synchronous approach. We analyze the impact of spatial variations of the clock signal, called clock skew, and temporal variations of the clock signal, called clock jitter, and introduce techniques to cope with it. These variations fundamentally limit the performance that can be achieved using a conventional design methodology.

At the other end of the design spectrum is an approach called asynchronous design, which avoids the problem of clock uncertainty altogether by eliminating the need for globally-distributed clocks. After discussing the basics of asynchronous design approach, we analyze the associated overhead and identify some practical applications. The important issue of synchronization, which is required when interfacing different clock domains or when sampling an asynchronous signal, also deserves some in-depth treatment. Finally, the fundamentals of on-chip clock generation using feedback is introduced along with trends in timing.

10.2 Classification of Digital Systems

In digital systems, signals can be classified depending on how they are related to a local clock [Messerschmitt90][Dally98]. Signals that transition only at predetermined periods in time can be classified as synchronous, mesochronous, or plesiochronous with respect to a system clock. A signal that can transition at arbitrary times is considered asynchronous.

10.2.1 Synchronous Interconnect

A synchronous signal is one that has the exact same frequency, and a known fixed phase offset with respect to the local clock. In such a timing methodology, the signal is “synchronized” with the clock, and the data can be sampled directly without any uncertainty. In digital logic design, synchronous systems are the most straightforward type of interconnect, where the flow of data in a circuit proceeds in lockstep with the system clock as shown below.

Here, the input data signal \( I_n \) is sampled with register \( R_1 \) to give signal \( C_{in} \), which is synchronous with the system clock and then passed along to the combinational logic block. After a suitable setting period, the output \( C_{out} \) becomes valid and can be sampled by
Section 10.2 Classification of Digital Systems

10.2.2 Mesochronous interconnect

A mesochronous signal is one that has the same frequency but an unknown phase offset with respect to the local clock (“meso” from Greek is middle). For example, if data is being passed between two different clock domains, then the data signal transmitted from the first module can have an unknown phase relationship to the clock of the receiving module. In such a system, it is not possible to directly sample the output at the receiving module because of the uncertainty in the phase offset. A (mesochronous) synchronizer can be used to synchronize the data signal with the receiving clock as shown below. The synchronizer serves to adjust the phase of the received signal to ensure proper sampling.

In Figure 10.2, signal $D_1$ is synchronous with respect to $Clk_A$. However, $D_1$ and $D_2$ are mesochronous with $Clk_B$ because of the unknown phase difference between $Clk_A$ and $Clk_B$ and the unknown interconnect delay in the path between Block A and Block B. The role of the synchronizer is to adjust the variable delay line such that the data signal $D_3$ (a delayed version of $D_2$) is aligned properly with the system clock of block B. In this example, the variable delay element is adjusted by measuring the phase difference between the received signal and the local clock. After register $R_2$ samples the incoming data during the certainty period, then signal $D_4$ becomes synchronous with $Clk_B$.

10.2.3 Plesiochronous Interconnect

A plesiochronous signal is one that has nominally the same, but slightly different frequency as the local clock (“plesio” from Greek is near). In effect, the phase difference
drifts in time. This scenario can easily arise when two interacting modules have independent clocks generated from separate crystal oscillators. Since the transmitted signal can arrive at the receiving module at a different rate than the local clock, one needs to utilize a buffering scheme to ensure all data is received. Typically, plesiochronous interconnect only occurs in distributed systems like long distance communications, since chip or even board level circuits typically utilize a common oscillator to derive local clocks. A possible framework for plesiochronous interconnect is shown in Figure 10.3.

In this digital communications framework, the originating module issues data at some unknown rate characterized by $C_1$, which is plesiochronous with respect to $C_2$. The timing recovery unit is responsible for deriving clock $C_3$ from the data sequence, and buffering the data in a FIFO. As a result, $C_3$ will be synchronous with the data at the input of the FIFO and will be mesochronous with $C_1$. Since the clock frequencies from the originating and receiving modules are mismatched, data might have to be dropped if the transmit frequency is faster, and data can be duplicated if the transmit frequency is slower than the receive frequency. However, by making the FIFO large enough, and periodically resetting the system whenever an overflow condition occurs, robust communication can be achieved.

### 10.2.4 Asynchronous Interconnect

Asynchronous signals can transition at any arbitrary time, and are not slaved to any local clock. As a result, it is not straightforward to map these arbitrary transitions into a synchronized data stream. Although it is possible to synchronize asynchronous signals by detecting events and introducing latencies into a data stream synchronized to a local clock, a more natural way to handle asynchronous signals is to simply eliminate the use of local clocks and utilize a self-timed asynchronous design approach. In such an approach, communication between modules is controlled through a handshaking protocol to perform the proper ordering of commands.

![Figure 10.4 Asynchronous design methodology for simple pipeline interconnect.](image-url)
When a logic block completes an operation, it will generate a completion signal $DV$ to indicate that output data is valid. The handshaking signals then initiate a data transfer to the next block, which latches in the new data and begins a new computation by asserting the initialization signal $I$. Asynchronous designs are advantageous because computations are performed at the native speed of the logic, where block computations occur whenever data becomes available. There is no need to manage clock skew, and the design methodology leads to a very modular approach where interaction between blocks simply occur through a handshaking procedure. However, these handshaking protocols result in increased complexity and overhead in communication that can reduce performance.

### 10.3 Synchronous Design — An In-depth Perspective

#### 10.3.1 Synchronous Timing Basics

Virtually all systems designed today use a periodic synchronization signal or clock. The generation and distribution of a clock has a significant impact on performance and power dissipation. For a positive edge-triggered system, the rising edge of the clock is used to denote the beginning and completion of a clock cycle. In the ideal world, assuming the clock paths from a central distribution point to each register are perfectly balanced, the phase of the clock (i.e., the position of the clock edge relative to a reference) at various points in the system is going to be exactly equal. However, the clock is neither perfectly periodic nor perfectly simultaneous. This results in performance degradation and/or circuit malfunction. Figure 10.5 shows the basic structure of a synchronous pipelined datapath. In the ideal scenario, the clock at registers 1 and 2 have the same clock period and transition at the exact same time. The following timing parameters characterize the timing of the sequential circuit.

- The contamination (minimum) delay $t_{c-q,cd}$ and maximum propagation delay of the register $t_{c-q}$.
- The set-up ($t_{su}$) and hold time ($t_{hold}$) for the registers.
- The contamination delay $t_{logic,cd}$ and maximum delay $t_{logic}$ of the combinational logic.

![Figure 10.5 Pipelined Datapath Circuit and timing parameters.](image-url)
TIMING ISSUES IN DIGITAL CIRCUITS  Chapter 10

- \( t_{i,1} \) and \( t_{i,2} \), corresponding to the position of the rising edge of the clock relative to a global reference.

Under ideal conditions \((t_{i,1} = t_{i,2})\), the worst case propagation delays determine the minimum clock period required for this sequential circuit. The period must be long enough for the data to propagate through the registers and logic and be set-up at the destination register before the next rising edge of the clock. This constraint is given by (as derived in Chapter 7):

\[
T > t_{c-q} + t_{\text{logic}} + t_{su}
\]  

(10.1)

At the same time, the hold time of the destination register must be shorter than the minimum propagation delay through the logic network,

\[
t_{\text{hold}} < t_{c-q, cd} + t_{\text{logic, cd}}
\]  

(10.2)

The above analysis is simplistic since the clock is never ideal. As a result of process and environmental variations, the clock signal can have spatial and temporal variations.

Clock Skew

The spatial variation in arrival time of a clock transition on an integrated circuit is commonly referred to as clock skew. The clock skew between two points \( i \) and \( j \) on a IC is given by \( \delta(i,j) = t_i - t_j \) where \( t_i \) and \( t_j \) are the position of the rising edge of the clock with respect to a reference. Consider the transfer of data between registers \( R_1 \) and \( R_2 \) in Figure 10.5. The clock skew can be positive or negative depending upon the routing direction and position of the clock source. The timing diagram for the case with positive skew is shown in Figure 10.6. As the figure illustrates, the rising clock edge is delayed by a positive \( \delta \) at the second register.

\[
\delta + t_h
\]

Figure 10.6 Timing diagram to study the impact of clock skew on performance and functionality. In this sample timing diagram, \( \delta > 0 \).

Clock skew is caused by static path-length mismatches in the clock load and by definition skew is constant from cycle to cycle. That is, if in one cycle \( CLK2 \) lagged \( CLK1 \) by \( \delta \), then on the next cycle it will lag it by the same amount. It is important to note that clock skew does not result in clock period variation, but rather phase shift.
Skew has strong implications on performance and functionality of a sequential system. First consider the impact of clock skew on performance. From Figure 10.6, a new input \( In \) sampled by \( R1 \) at edge 1 will propagate through the combinational logic and be sampled by \( R2 \) on edge 4. If the clock skew is positive, the time available for signal to propagate from \( R1 \) to \( R2 \) is increased by the skew \( \delta \). The output of the combinational logic must be valid one set-up time before the rising edge of \( CLK2 \) (point 4). The constraint on the minimum clock period can then be derived as:

\[
T + \delta \geq t_{c-q} + t_{\text{logic}} + t_{su} \quad \text{or} \quad T \geq t_{c-q} + t_{\text{logic}} + t_{su} - \delta
\]  

(10.3)

The above equation suggests that clock skew actually has the potential to improve the performance of the circuit. That is, the minimum clock period required to operate the circuit reliably reduces with increasing clock skew! This is indeed correct, but unfortunately, increasing skew makes the circuit more susceptible to race conditions may and harm the correct operation of sequential systems.

As above, assume that input \( In \) is sampled on the rising edge of \( CLK1 \) at edge 1 into \( R1 \). The new values at the output of \( R1 \) propagates through the combinational logic and should be valid before edge 2 at \( CLK2 \). However, if the minimum delay of the combinational logic block is small, the inputs to \( R2 \) may change before the clock edge 2, resulting in incorrect evaluation. To avoid races, we must ensure that the minimum propagation delay through the register and logic must be long enough such that the inputs to \( R2 \) are valid for a hold time after edge 2. The constraint can be formally stated as:

\[
\delta + t_{\text{hold}} < t_{c-q, cd} + t_{\text{logic, cd}}
\]

or

\[
\delta < t_{c-q, cd} + t_{\text{logic, cd}} - t_{\text{hold}}
\]

(10.4)

Figure 10.7 shows the timing diagram for the case when \( \delta < 0 \). For this case, the rising edge of \( CLK2 \) happens before the rising edge of \( CLK1 \). On the rising edge of \( CLK1 \), a new input is sampled by \( R1 \). The new sampled data propagates through the combinational logic and is sampled by \( R2 \) on the rising edge of \( CLK2 \), which corresponds to edge 4. As can be seen from Figure 10.7 and Eq. (10.3), a negative skew directly impacts the performance of sequential system. However, a negative skew implies that the system never fails, since edge 2 happens before edge 1! This can also be seen from Eq. (10.4), which is always satisfied since \( \delta < 0 \).
Example scenarios for positive and negative clock skew are shown in Figure 10.8.

- $\delta > 0$—This corresponds to a clock routed in the same direction as the flow of the data through the pipeline (Figure 10.8a). In this case, the skew has to be strictly controlled and satisfy Eq. (10.4). If this constraint is not met, the circuit does malfunction independent of the clock period. Reducing the clock frequency of an edge-triggered circuit does not help get around skew problems! On the other hand, positive skew increases the throughput of the circuit as expressed by Eq. (10.3), because the clock period can be shortened by $\delta$. The extent of this improvement is limited as large values of $\delta$ soon provoke violations of Eq. (10.4).

- $\delta < 0$—When the clock is routed in the opposite direction of the data (Figure 10.8b), the skew is negative and condition (10.4) is unconditionally met. The circuit operates correctly independent of the skew. The skew reduces the time available for actual computation so that the clock period has to be increased by $|\delta|$. In summary, routing the clock in the opposite direction of the data avoids disasters but hampers the circuit performance.

Unfortunately, since a general logic circuit can have data flowing in both directions (for example, circuits with feedback), this solution to eliminate races will not always work (Figure 10.9). The skew can assume both positive and negative values depending on the direction of...
the data transfer. Under these circumstances, the designer has to account for the worst-case skew condition. In general, routing the clock so that only negative skew occurs is not feasible. Therefore, the design of a low-skew clock network is essential.

Example 10.1 Propagation and Contamination Delay Estimation

Consider the logic network shown in Figure 10.10. Determine the propagation and contamination delay of the network, assuming that the worst case gate delay is $t_{\text{gate}}$. The maximum and minimum delays of the gates is made, as they are assumed to be identical.

The contamination delay is given by $2 t_{\text{gate}}$ (the delay through $OR_1$ and $OR_2$). On the other hand, computation of the worst case propagation delay is not as simple as it appears. At first glance, it would appear that the worst case corresponds to path $\oplus$ and the delay is $5 t_{\text{gate}}$. However, when analyzing the data dependencies, it becomes obvious that path $\oplus$ is never exercised. Path $\oplus$ is called a false path. If $A = 1$, the critical path goes through $OR_1$ and $OR_2$. If $A = 0$ and $B = 0$, the critical path is through $I_1, OR_1$ and $OR_2$ (corresponding to a delay of $3 t_{\text{gate}}$). For the case when $A = 0$ and $B = 1$, the critical path is through $I_1, OR_1, AND_3$ and $OR_2$. In other words, for this simple (but contrived) logic circuit, the output does not even depend on inputs $C$ and $D$ (that is, there is redundancy). Therefore, the propagation delay is $4 t_{\text{gate}}$. Given the propagation and contamination delay, the minimum and maximum allowable skew can be easily computed.

![Figure 10.10 Logic network for computation of performance.](image)

**WARNING:** The computation of the worst-case propagation delay for combinational logic, due to the existence of false paths, cannot be obtained by simply adding the propagation delay of individual logic gates. The critical path is strongly dependent on circuit topology and data dependencies.

Clock Jitter

*Clock jitter* refers to the temporal variation of the clock period at a given point — that is, the clock period can reduce or expand on a cycle-by-cycle basis. It is strictly a temporal uncertainty measure and is often specified at a given point on the chip. Jitter can be measured and cited in one of many ways. *Cycle-to-cycle jitter* refers to time varying deviation
of a single clock period and for a given spatial location \( i \) is given as

\[
T_{\text{jitter}, i}(n) = T_{i,n+1} - T_{i,n} - T_{\text{CLK}},
\]

where \( T_{i,n} \) is the clock period for period \( n \), \( T_{i,n+1} \) is clock period for period \( n+1 \), and \( T_{\text{CLK}} \) is the nominal clock period.

Jitter directly impacts the performance of a sequential system. Figure 10.11 shows the nominal clock period as well as variation in period. Ideally the clock period starts at edge 2 and ends at edge 5 and with a nominal clock period of \( T_{\text{CLK}} \). However, as a result of jitter, the worst case scenario happens when the leading edge of the current clock period is delayed (edge 3), and the leading edge of the next clock period occurs early (edge 4). As a result, the total time available to complete the operation is reduced by 2 \( t_{\text{jitter}} \) in the worst case and is given by

\[
T_{\text{CLK}} - 2t_{\text{jitter}} \geq t_{c-q} + t_{\text{logic}} + t_{su} \quad \text{or} \quad T \geq t_{c-q} + t_{\text{logic}} + t_{su} + 2t_{\text{jitter}}
\]

The above equation illustrates that jitter directly reduces the performance of a sequential circuit. Care must be taken to reduce jitter in the clock network to maximize performance.

**Impact of Skew and Jitter on Performance**

In this section, the combined impact of skew and jitter is studied with respect to conventional edge-triggered clocking. Consider the sequential circuit shown in Figure 10.12.

Assume that nominally ideal clocks are distributed to both registers (the clock period is identical every cycle and the skew is 0). In reality, there is static skew \( \delta \) between the two clock signals (assume that \( \delta > 0 \)). Assume that \( \text{CLK1} \) has a jitter of \( t_{\text{jitter1}} \) and \( \text{CLK2} \) has a jitter of \( t_{\text{jitter2}} \). To determine the constraint on the minimum clock period, we must look at the minimum available time to perform the required computation. The worst case happen when the leading edge of the current clock period on \( \text{CLK1} \) happens late (edge 3) and the leading edge of the next cycle of \( \text{CLK2} \) happens early (edge 5). This results in the following constraint

\[
T_{\text{CLK}} + \delta - t_{\text{jitter1}} - t_{\text{jitter2}} \geq t_{c-q} + t_{\text{logic}} + t_{su}
\]

or

\[
T \geq t_{c-q} + t_{\text{logic}} + t_{su} - \delta + t_{\text{jitter1}} + t_{\text{jitter2}}
\]

(10.6)
As the above equation illustrates, while positive skew can provide potential performance advantage, jitter has a negative impact on the minimum clock period. To formulate the minimum delay constraint, consider the case when the leading edge of the \( CLK1 \) cycle arrives early (edge 1) and the leading edge the current cycle of \( CLK2 \) arrives late (edge 6). The separation between edge 1 and 6 should be smaller than the minimum delay through the network. This results in

\[
\delta + t_{\text{hold}} + t_{\text{jitter 1}} + t_{\text{jitter 2}} < t_{(c-q, cd)} + t_{(\text{logic, cd)}}
\]

or

\[
\delta < t_{(c-q, cd)} + t_{(\text{logic, cd)}} - t_{\text{hold}} - t_{\text{jitter 1}} - t_{\text{jitter 2}}
\]  

The above relation indicates that the acceptable skew is reduced by the jitter of the two signals.

Now consider the case when the skew is negative (\( \delta < 0 \)) as shown in Figure 10.13. For the timing shown, \( |\delta| > t_{\text{jitter 2}} \). It can be easily verified that the worst case timing is exactly the same as the previous analysis, with \( \delta \) taking a negative value. That is, negative skew reduces performance.

\[\text{Figure 10.12} \text{ Sequential circuit to study the impact of skew and jitter on edge-triggered systems. In this example, a positive skew (\( \delta \)) is assumed.}\]

\[\text{Figure 10.13} \text{ Consider a negative clock skew (\( \delta \)) and the skew is assumed to be larger than the jitter.}\]
10.3.2 Sources of Skew and Jitter

A perfect clock is defined as perfectly periodic signal that is simultaneous triggered at various memory elements on the chip. However, due to a variety of process and environmental variations, clocks are not ideal. To illustrate the sources of skew and jitter, consider the simplistic view of clock generation and distribution as shown in Figure 10.14. Typically, a high frequency clock is either provided from off chip or generated on-chip. From a central point, the clock is distributed using multiple matched paths to low-level memory elements registers. In this picture, two paths are shown. The clock paths include wiring and the associated distributed buffers required to drive interconnects and loads. A key point to realize in clock distribution is that the absolute delay through a clock distribution path is not important; what matters is the relative arrival time between the output of each path at the register points (i.e., it is perfectly acceptable for the clock signal to take multiple cycles to get from a central distribution point to a low-level register as long as all clocks arrive at the same time to different registers on the chip).

![Figure 10.14 Skew and jitter sources in synchronous clock distribution.](image)

The are many reasons why the two parallel paths don’t result in exactly the same delay. The sources of clock uncertainty can be classified in several ways. First, errors can be divided into systematic or random. Systematic errors are nominally identical from chip to chip, and are typically predictable (e.g., variation in total load capacitance of each clock path). In principle, such errors can be modeled and corrected at design time given sufficiently good models and simulators. Failing that, systematic errors can be deduced from measurements over a set of chips, and the design adjusted to compensate. Random errors are due to manufacturing variations (e.g., dopant fluctuations that result in threshold variations) that are difficult to model and eliminate. Mismatch may also be characterized as static or time-varying. In practice, there is a continuum between changes that are slower than the time constant of interest, and those that are faster. For example, temperature variations on a chip vary on a millisecond time scale. A clock network tuned by a one-time calibration or trimming would be vulnerable to time-varying mismatch due to varying thermal gradients. On the other hand, to a feedback network with a bandwidth of several megahertz, thermal changes appear essentially static. For example, the clock net is usually by far the largest single net on the chip, and simultaneous transitions on the clock drivers induces noise on the power supply. However, this high speed effect does not contribute to
time-varying mismatch because it is the same on every clock cycle, affecting each rising clock edge the same way. Of course, this power supply glitch may still cause static mismatch if it is not the same throughout the chip. Below, the various sources of skew and jitter, introduced in Figure 10.14, are described in detail.

**Clock-Signal Generation** (1)

The generation of the clock signal itself causes jitter. A typical on-chip clock generator, as described at the end of this chapter, takes a low-frequency reference clock signal, and produces a high-frequency global reference for the processor. The core of such a generator is a Voltage-Controlled Oscillator (VCO). This is an analog circuit, sensitive to intrinsic device noise and power supply variations. A major problem is the coupling from the surrounding noisy digital circuitry through the substrate. This is particularly a problem in modern fabrication processes that combine a lightly-doped epitaxial layer and a heavily-doped substrate (to combat latch-up). This causes substrate noise to travel over large distances on the chip. These noise sources cause temporal variations of the clock signal that propagate unfiltered through the clock drivers to the flip-flops, and result in cycle-to-cycle clock-period variations. This jitter causes performance degradation.

**Manufacturing Device Variations** (2)

Distributed buffers are integral components of the clock distribution networks, as they are required to drive both the register loads as well as the global and local interconnects. The matching of devices in the buffers along multiple clock paths is critical to minimizing timing uncertainty. Unfortunately, as a result of process variations, devices parameters in the buffers vary along different paths, resulting in static skew. There are many sources of variations including oxide variations (that affects the gain and threshold), dopant variations, and lateral dimension (width and length) variations. The doping variations can affect the depth of junction and dopant profiles and cause variations in electrical parameters such as device threshold and parasitic capacitances. The orientation of polysilicon can also have a big impact on the device parameters. Keeping the orientation the same across the chip for the clock drivers is critical.

Variation in the polysilicon critical dimension, is particularly important as it translates directly into MOS transistor channel length variation and resulting variations in the drive current and switching characteristics. Spatial variation usually consists of wafer-level (or within-wafer) variation and die-level (or within-die) variation. At least part of the variation is systematic and can be modeled and compensated for. The random variations however, ultimately limits the matching and skew that can be achieved.
Interconnect Variations

Vertical and lateral dimension variations cause the interconnect capacitance and resistance to vary across a chip. Since this variation is static, it causes skew between different paths. One important source of interconnect variation is the Inter-level Dielectric (ILD) thickness variations. In the formation of aluminum interconnect, layers of silicon dioxide are interposed between layers of patterned metallization. The oxide layer is deposited over a layer of patterned metal features, generally resulting in some remaining step height or surface topography. Chemical-mechanical polishing (CMP) is used to “planarize” the surface and remove topography resulting from deposition and etch (as shown in Figure 10.15a). While at the feature scale (over an individual metal line), CMP can achieve excellent planarity, there are limitations on the planarization that can be achieved over a global range. This is primarily caused due to variations in polish rate that is a function of the circuit layout density and pattern effects. Figure 10.15b shows this effect where the polish rate is higher for the lower spatial density region, resulting in smaller dielectric thickness and higher capacitance.

The assessment and control of variation is of critical importance in semiconductor process development and manufacturing. Significant advances have been made to develop analytical models for estimating the ILD thickness variations based on spatial density. Since this component is often predictable from the layout, it is possible to actually correct for the systematic component at design time (e.g., by adding appropriate delays or making the density uniform by adding “dummy fills”). Figure 10.16 shows the spatial pattern density and ILD thickness for a high performance microprocessor. The graphs show that there is clear correlation between the density and the thickness of the dielectric. So clock distribution networks must exploit such information to reduce clock skew.

Other interconnect variations include deviation in the width of the wires and line spacing. This results from photolithography and etch dependencies. At the lower levels of metallization, lithographic effects are important while at higher levels etch effects are important that depend on width and layout. The width is a critical parameter as it directly impacts the resistance of the line and the wire spacing affects the wire-to-wire capacitance.

![Figure 10.15](image-url) Inter-level Dielectric (ILD) thickness variation due to density (courtesy of Duane Boning).
Section 10.3 Synchronous Design — An In-depth Perspective

Environmental Variations (4 and 5)

Environmental variations are probably the most significant and primarily contribute to skew and jitter. The two major sources of environmental variations are temperature and power supply. Temperature gradients across the chip is a result of variations in power dissipation across the die. This has particularly become an issue with clock gating where some parts of the chip maybe idle while other parts of the chip might be fully active. This results in large temperature variations. Since the device parameters (such as threshold, mobility, etc.) depend strongly on temperature, buffer delay for a clock distribution network along one path can vary drastically for another path. More importantly, this component is time-varying since the temperature changes as the logic activity of the circuit varies. As a result, it is not sufficient to simulate the clock networks at worst case corners of temperature; instead, the worst-case variation in temperature must be simulated. An interesting question is does temperature variation contribute to skew or to jitter? Clearly the variation in temperature is time varying but the changes are relatively slow (typical time constants for temperature on the order of milliseconds). Therefore it usually considered as a skew component and the worst-case conditions are used. Fortunately, using feedback, it is possible to calibrate the temperature and compensate.

Power supply variations on the hand is the major source of jitter in clock distribution networks. The delay through buffers is a very strong function of power supply as it directly affects the drive of the transistors. As with temperature, the power supply voltage is a strong function of the switching activity. Therefore, the buffer delay along one path is very different than the buffer delay along another path. Power supply variations can be classified into static (or slow) and high frequency variations. Static power supply variations may result from fixed currents drawn from various modules, while high-frequency variations result from instantaneous IR drops along the power grid due to fluctuations in switching activity. Inductive issues on the power supply are also a major concern since they cause voltage fluctuations. This has particularly become a concern with clock gating as the load current can vary dramatically as the logic transitions back and forth between the idle and active states. Since the power supply can change rapidly, the period of the
clock signal is modulated on a cycle-by-cycle basis, resulting in _jitter_. The _jitter_ on two different clock points may be correlated or uncorrelated depending on how the power network is configured and the profile of switching patterns. Unfortunately, high-frequency power supply changes are difficult to compensate even with feedback techniques. As a result, power supply noise fundamentally limits the performance of clock networks.

**Capacitive Coupling (X and X)**

The variation in capacitive load also contributes to timing uncertainty. There are two major sources of capacitive load variations: coupling between the clock lines and adjacent signal wires and variation in gate capacitance. The clock network includes both the interconnect and the gate capacitance of latches and registers. Any coupling between the clock wire and adjacent signal results in timing uncertainty. Since the adjacent signal can transition in arbitrary directions and at arbitrary times, the exactly coupling to the clock network is not fixed from cycle-to-cycle. This results in clock _jitter_. Another major source of clock uncertainty is variation in the gate capacitance related to the sequential elements. The load capacitance is highly non-linear and depends on the applied voltage. In many latches and registers this translates to the clock load being a function of the current state of the latch/register (this is, the values stored on the internal nodes of the circuit), as well as the next state. This causes the delay through the clock buffers to vary from cycle-to-cycle, causing _jitter_.

**Example 10.2 Data-dependent Clock Jitter**

Consider the circuit shown in Figure 10.17, where a minimum-sized local clock buffer drives a register (actually, four registers are driven, though only one is shown here). The simulation shows _CKb_, the output of the first inverter for four possible transitions (0→0, 0→1, 1→0 and 1→1). The jitter on the clock based on data-dependent capacitance is illustrated. In general, the only way to deal with this problem is to use registers that don’t exhibit a large variation in load as a function of data — for example, the differential sense-amplifier register shown in Chapter 7.

![Figure 10.17 Impact of data-dependent clock load on clock jitter for pass-transistor register.](image_url)
10.3.3 Clock-Distribution Techniques

It is clear from the previous discussion that clock skew and jitter are major issues in digital circuits, and can fundamentally limit the performance of a digital system. It is necessary to design a clock network that minimizes skew and jitter. Another important consideration in clock distribution is the power dissipation. In most high-speed digital processors, a majority of the power is dissipated in the clock network. To reduce power dissipation, clock networks must support clock conditioning — this is, the ability to shut down parts of the clock network. Unfortunately, clock gating results in additional clock uncertainty.

In this section, an overview of basic constructs in high-performance clock distribution techniques is presented along with a case study of clock distribution in the Alpha microprocessor. There are many degrees of freedom in the design of a clock network including the type of material used for wires, the basic topology and hierarchy, the sizing of wires and buffers, the rise and fall times, and the partitioning of load capacitances.

Fabrics for clocking

Clock networks typically include a network that is used to distribute a global reference to various parts of the chip, and a final stage that is responsible for local distribution of the clock while considering the local load variations. Most clock distribution schemes exploit the fact that the absolute delay from a central clock source to the clocking elements is irrelevant — only the relative phase between two clocking points is important. Therefore one common approach to distributing a clock is to use balanced paths (or called trees).

The most common type of clock primitive is the H-tree network (named for the physical structure of the network) as illustrated in Figure 10.18, where a 4x4 array is shown. In this scheme, the clock is routed to a central point on the chip and balanced paths, that include both matched interconnect as well as buffers, are used to distribute the reference to various leaf nodes. Ideally, if each path is balanced, the clock skew is zero. That is, though it might take multiple clock cycles for a signal to propagate from the central point to each leaf node, the arrival times are equal at every leaf node. However, in reality, as discussed in the previous section, process and environmental variations cause clock skew and jitter to occur.

Figure 10.18 Example of an H-tree clock-distribution network for 16 leaf nodes.

The H-tree configuration is particularly useful for regular-array networks in which all elements are identical and the clock can be distributed as a binary tree (for example, arrays of identical tiled processors). The concept can be generalized to a more generic set-
The more general approach, referred to as routed RC trees, represents a floorplan that distributes the clock signal so that the interconnections carrying the clock signals to the functional sub-blocks are of equal length. That is, the general approach does not rely on a regular physical structure. An example of a matched RC is shown in Figure 10.19. The chip is partitioned into ten balanced load segments (tiles). The global clock driver distributes the clock to tile drivers located at the dots in the figure. A lower level RC-matched tree is used to drive 580 additional drivers inside each tile.

Another important clock distribution template is the grid structure as shown in Figure 10.20 [Bailey00]. Grids are typically used in the final stage of clock network to distribute the clock to the clocking element loads. This approach is fundamentally different from the balanced RC approach. The main difference is that the delay from the final driver to each load is not matched. Rather, the absolute delay is minimized assuming that the grid size is small. A major advantage of such a grid structure is that it allows for late design changes since the clock is easily accessible at various points on the die. Unfortunately, the penalty is the power dissipation since the structure has a lot of unnecessary interconnect.

It is essential to consider clock distribution in the earlier phases of the design of a complex circuit, since it might influence the shape and form of the chip floorplan. Clock distribution is often only considered in the last phases of the design process, when most of the chip layout is already frozen. This results in unwieldy clock networks and multiple
Section 10.3  Synchronous Design — An In-depth Perspective

Timing constraints that hamper the performance and operation of the final circuit. With careful planning, a designer can avoid many of these problems, and clock distribution becomes a manageable operation.

Case Study—The Digital Alpha Microprocessors

In this section, the clock distribution strategy for three generations of the Alpha microprocessor is discussed in detail. These processors have always been at the cutting edge of the technology, and therefore represent an interesting perspective on the evolution of clock distribution.

The Alpha 21064 Processor. The first generation Alpha microprocessor (21064 or EV4) from Digital Equipment Corporation used a single global clock driver [Dobberpuhl92]. The distribution of clock load capacitance among various functional block shown in Figure 10.20, resulting in a total clock load of 3.25nF! The processor uses a single-phase clock methodology and the 200Mhz clock is fed to a binary fanning tree with five levels of buffering. The inputs to the clock drivers are shorted out to smooth out the asymmetry in the incoming signals. The final output stage, residing in the middle of the chip, drives the clock net. The clock driver and the associated pre-drivers account for 40% of the effective switched capacitance (12.5nF), resulting in significant power dissipation. The overall width of the clock driver was on the order of 35cm in a 0.75 µm technology. A detail clock skew simulation with process variations indicates that a clock uncertainty of less than 200psec (< 10%) was achieved.

The Alpha 21164 Processor. The Alpha 21164 microprocessor (EV5) operates at a clock frequency of 300 Mhz while using 9.3 million transistors on a 16.5 mm × 18.1 mm die in a 0.5 µm CMOS technology [Bowhill95]. A single-phase clocking methodology was selected and the design made extensive use of dynamic logic, resulting in a substantial clock load of 3.75 nF. The clock-distribution system consumes 20 W, which is 40% of the total dissipation of the processor.

The incoming clock signal is first routed through a single six-stage buffer placed at the center of the chip. The resulting signal is distributed in metal-3 to the left and right banks of final clock drivers, positioned between the secondary cache memory and the outside edge of the execution unit (Figure 10.22a). The produced clock signal is driven onto a grid of metal-3 and metal-4 wires. The equivalent transistor width of the final driver inverter equals 58 cm! To ensure the integrity of the clock grid across the chip, the grid was extracted from the layout, and the resulting RC-network was simulated. A three-dimensional representation of the simulation results is plotted in Figure 10.22b. As evi-
dent from the plot, the skew is zero at the output of the left and right drivers. The maximum value of the absolute skew is smaller than 90 psec. The critical instruction and execution units all see the clock within 65 psec.

Clock skew and race problems were addressed using a mix-and-match approach. The clock skew problems were eliminated by either routing the clock in the opposite direction of the data at a small expense of performance or by ensuring that the data could not overtake the clock. A standardized library of level-sensitive transmission-gate latches was used for the complete chip. To avoid race-through conditions, a number of design guidelines were followed including:

- Careful sizing of the local clock buffers so that their skew was minimal.
- At least one gate had to be inserted between connecting latches. This gate, which can be part of the logic function or just a simple inverter, ensures that the signal cannot overtake the clock. Special design verification tools were developed to guarantee that this rule was obeyed over the complete chip.

To improve the inter-layer dielectric uniformity, filler polygons were inserted between widely spaced lines (Figure 10.23). Though this may increase the capacitance to nearby signal lines, the improved uniformity results in lower variation and clock uncertainty. The dummy fills are automatically inserted, and tied to one of the power rails ($V_{DD}$ or $GND$). This technique is used in many processors for controlling the clock skew.

This example demonstrates that managing clock skew and clock distribution for large, high-performance synchro-
Section 10.3  Synchronous Design — An In-depth Perspective

nous designs is a feasible task. However, making such a circuit work in a reliable way requires careful planning and intensive analysis.

The Alpha 21264 Processor. A hierarchical clocking scheme is used in the 600Mhz Alpha 21264 (EV6) processor (in 0.35µm CMOS) as shown in Figure 10.24. The choice of a hierarchical clocking scheme for this processor is significantly different than their previous approaches, which did not have a hierarchy of clocks beyond the global clock grid. Using a hierarchical clocking approach makes trade-off’s between power and skew management. Power is reduced because the clocking networks for individual blocks can be gated. As seen in previous generation microprocessors, the clock power contributes to a large fraction of overall power consumption. Also the flexibility of having local clocks provides the designers with more freedom with circuit styles at the module level. The drawback of using a hierarchical clock network is that skew reduction becomes more difficult because clocks to various local registers may go through very different paths, which may contribute to the skew. However, using timing verification tools, the skew can be managed by tweaking of clock drivers.

The clock hierarchy consists of a global clock grid, called GCLK, that covers the entire die. State elements and clocking points exist from 0 to 8 levels past GCLK. The on-chip generated clock is routed to the center of the die and distributed using tree structures to 16 distributed clock drivers (Figure 10.25). The global clock distribution network utilizes a windowpane configuration, that achieves low skew by dividing up the clock into 4 regions -- this reduces the distance from the drivers to the loads. Each grid pane is driven from four sides, reducing the dependence on process variations. This also helps the power supply and thermal problems as the drivers are distributed through the chip.

Use of a grid-ded clock has the advantage of reducing the clock skew and provides universal availability of clock signals. The drawback clearly is the increased capacitance of the Global Clock grid when compared to a tree distribution approach. Beyond the GCLK is a major clock grid. The major clock grid are used to drive different large execution blocks within the chip including 1) Bus interface unit 2) integer issue and execution units 3) floating point issue and execution units 4) instruction fetch and branch prediction unit 5) load/store unit and 6) pad ring. The reason the major clocks were introduced was to reduce power because the major clocks have localized loads, and can be sized appropriately to meet the skew and edge requirements for the local loading conditions.

Figure 10.24 Clock hierarchy for the Alpha 21264 Processor.
TIMING ISSUES IN DIGITAL CIRCUITS

Chapter 10

The last hierarchy of clocks in the Alpha processor are the local clocks, which are generated as needed from any other clock, and typically can be customized to meet local timing constraints. The local clocks provide great flexibility in the design of the local logic blocks, but at the same time, makes it significantly more difficult to manage skew. For example, local clocks can be gated in order to reduce power, can be driven with wildly different loads, and can even be tweaked to enable time borrowing between different blocks. Furthermore, the local clocks are susceptible to coupling from data lines as well because they are local and not shielded like the global grid-dec clocks. As a result, the local clock distribution is highly dependent on it’s local interconnection, and has to be designed very carefully to manage local clock skew.

To fully exploit the improved performance of logic gates with technology scaling, clock skew and jitter must be carefully addressed. Skew and jitter can fundamentally limit the performance of a digital circuits. Some guidelines for reducing of clock skew and jitter are presented below.

1. To minimize skew, balance clock paths from a central distribution source to individual clocking elements using H-tree structures or more generally routed tree structures. When using routed clock trees, the effective clock load of each path that includes wiring as well as transistor loads must be equalized.

2. The use of local clock grids (instead of routed trees) can reduce skew at the cost of increased capacitive load and power dissipation.

3. If data dependent clock load variations causes significant jitter, differential registers that have a data independent clock load should be used. The use of gated clocks to save also results in data dependent clock load and increased jitter. In clock networks where the fixed load is large (e.g., using clock grids), the data dependent variation might not be significant.

4. If data flows in one direction, route data and clock in opposite directions. This eliminates races at the cost of performance.
5. Avoid data dependent noise by shielding clock wires from adjacent signal wires. By placing power lines (V_{DD} or GND) next to the clock wires, coupling from neighboring signal nets can be minimized or avoided.

6. Variations in interconnect capacitance due to inter-layer dielectric thickness variation can be greatly reduced through the use of dummy fills. Dummy fills are very common and reduce skew by increasing uniformity. Systematic variations should be modeled and compensated for.

7. Variation in chip temperature across the die causes variations in clock buffer delay. The use of feedback circuits based on delay locked loops as discussed later in this chapter can easily compensate for temperature variations.

8. Power supply variation is a significant component of jitter as it impacts the cycle to cycle delay through clock buffers. High frequency power supply variation can be reduced by addition of on-chip decoupling capacitors. Unfortunately, decoupling capacitors require a significant amount of area and efficient packaging solutions must be leveraged to reduce chip area.

### 10.3.4 Latch-Based Clocking

While the use of registers in a sequential circuits enables a robust design methodology, there are significant performance advantages to using a latch based design in which combinational logic is separated by transparent latches. In an edge-triggered system, the worst case logic path between two registers determines the minimum clock period for the entire system. If a logic block finishes before the clock period, it has to idle till the next input is latched in on the next system clock edge. The use of a latch based methodology (as illustrated in Figure 10.26) enables more flexible timing, allowing one stage to pass slack to or steal time from following stages. This flexibility, allows an overall performance increase. Note that the latch based methodology is nothing more than adding logic between latches of a master-slave flip-flop.

For the latch-based system in Figure 10.26, assume a two-phase clocking scheme. Assume furthermore that the clock are ideal, and that the two clocks are inverted versions of each other (for sake of simplicity). In this configuration, a stable input is available to the combinational logic block A (CLB_A) on the falling edge of CLK1 (at edge Î²) and it has a maximum time equal to the \( T_{CLK} \) to evaluate (that is, the entire low phase of CLK1). On the falling edge of CLK2 (at edge Î³), the output CLB_A is latched and the computation of CLK_B is launched. CLB_B computes on the low phase of CLK2 and the output is available on the falling edge of CLK1 (at edge Î©). This timing appears equivalent to having an edge-triggered system where CLB_A and CLB_B are cascaded and between two edge-triggered registers (Figure 10.27). In both cases, it appears that the time available to perform the combination of CLB_A and CLB_B is \( T_{CLK} \).
However, there is an important performance related difference. In a latch based system, since the logic is separated by level sensitive latches, it is possible for a logic block to utilize time that is left over from the previous logic block and this is referred to as *slack borrowing* [Bernstein00]. This approach requires no explicit design changes, as the passing of slack from one block to the next is automatic. The key advantage of slack borrowing is that it allows logic between cycle boundaries to use more than one clock cycle while satisfying the cycle time constraint. Stated in another way, if the sequential system works at a particular clock rate and the total logic delay for a complete cycle is larger than the clock period, then unused time or *slack* has been implicitly borrowed from preceding stages. This implies that the clock rate can be higher than the worst case critical path!

Slack passing happens due to the level sensitive nature of latches. In Figure 10.26, the input to $CLB_A$ should be valid by the falling edge of CLK1 (edge $\oplus$). What happens if the combinational logic block of the previous stage finishes early and has a valid input data for $CLB_A$ before edge $\oplus$? Since the a latch is transparent during the entire high phase of the clock, as soon as the previous stage has finished computing, the input data for $CLB_A$ is valid. This implies that the maximum time available for $CLB_A$ is its phase time (i.e., the low phase of CLK1) and any left over time from the previous computation. Formally state, slack passing has taken place if $T_{CLK} < t_{pd,A} + t_{pd,B}$ and the logic functions correctly (for simplicity, the delay associated with latches are ignored). An excellent quantitative analysis of slack-borrowing is presented in [Bernstein00].

Consider the latch based system in Figure 10.28. In this example, point $a$ (input to $CLB_A$) is valid before the edge $\oplus$. This implies that the previous block did not use up the entire high phase of CLK1, which results in slack time (denoted by the shaded area). By
construction, $CLB_A$ can start computing as soon as point $a$ becomes valid and uses the slack time to finish well before its allocated time (edge $\square$). Since $L2$ is a transparent latch, $c$ becomes valid on the high phase of $CLK2$ and $CLB_B$ starts to compute by using the slack provided by $CLB_A$. $CLB_B$ completes before its allocated time (edge $\triangle$) and passes a small slack to the next cycle. As this picture indicates, the total cycle delay, that is the sum of the delay for $CLB_A$ and $CLB_B$, is larger than the clock period. Since the pipeline behaves correctly, slack passing has taken place and a higher throughput has been achieved.

An important question related to slack passing relates to the maximum possible slack that can be passed across cycle boundaries. In Figure 10.28, it is easy to see that the earliest time that $CLB_A$ can start computing is $\Box$. This happens if the previous logic block did not use any of its allocated time ($CLK1$ high phase) or if it finished by using slack from previous stages. Therefore, the maximum time that can be borrowed from the previous stage is $1/2$ cycle or $T_{CLK}/2$. Similarly, $CLB_B$ must finish its operation by edge $\triangle$. This implies that the maximum logic cycle delay is equal to $1.5 \times T_{CLK}$. However, note that for an $n$-stage pipeline, the overall logic delay cannot exceed the time available of $n \times T_{CLK}$.

**Example 10.3 Slack-passing example**

First consider an edge-triggered pipeline of Figure 10.29. Assume that the primary input $In$ is valid slightly after the rising edge of the clock. For this circuit, it is easy to see that the minimum clock period required is 125ns. The latency is two clock cycles (actually, the output is valid 2.5 cycles after the input settles). Note that for the first pipeline stage, $1/2$
cycle is wasted as the input data can only be latched in on the falling edge of the clock. This time can be exploited in a latch based system.

Figure 10.30 shows a latch based pipeline of the same sequential circuit. As the timing indicates, exactly the same timing can be achieve with a clock period of 100ns. This is enabled by slack borrowing between logical partitions.

10.4 Self-Timed Circuit Design*

10.4.1 Self-Timed Logic - An Asynchronous Technique

The synchronous design approach advocated in the previous sections assumes that all circuit events are orchestrated by a central clock. Those clocks have a dual function.

- They insure that the physical timing constraints are met. The next clock cycle can only start when all logic transitions have settled and the system has come to a steady state. This ensures that only legal logical values are applied in the next round of computation. In short, clocks account for the worst case delays of logic gates, sequential logic elements and the wiring.

- Clock events serve as a logical ordering mechanism for the global system events. A clock provides a time base that determines what will happen and when. On every
clock transition, a number of operations are initiated that change the state of the sequential network.

Consider the pipelined datapath of Figure 10.31. In this circuit, the data transitions through logic stages under the command of the clock. The important point to note under this methodology is that the clock period is chosen to be larger than the worst-case delay of each pipeline stage, or \( T > \max(t_{pd1}, t_{pd2}, t_{pd3}) + t_{pd,\text{reg}} \). This will ensure satisfying the physical constraint. At each clock transition, a new set of inputs is sampled and computation is started anew. The throughput of the system—which is equivalent to the number of data samples processed per second—is equivalent to the clock rate. When to sample a new input or when an output is available depends upon the logical ordering of the system events and is clearly orchestrated by the clock in this example.

The synchronous design methodology has some clear advantages. It presents a structured, deterministic approach to the problem of choreographing the myriad of events that take place in digital designs. The approach taken is to equalize the delays of all operations by making them as bad as the worst of the set. The approach is robust and easy to adhere to, which explains its enormous popularity; however it does have some pitfalls.

- It assumes that all clock events or timing references happen simultaneously over the complete circuit. This is not the case in reality, because of effects such as clock skew and jitter.

- As all the clocks in a circuit transitions at the same time, significant current flows over a very short period of time (due to the large capacitance load). This causes significant noise problems due to package inductance and power supply grid resistance.

- The linking of physical and logical constraints has some obvious effects on the performance. For instance, the throughput rate of the pipelined system of Figure 10.31 is directly linked to the worst-case delay of the slowest element in the pipeline. On the average, the delay of each pipeline stage is smaller. The same pipeline could support an average throughput rate that is substantially higher than the synchronous one. For example, the propagation delay of a 16-bit adder is highly data dependent (e.g., adding two 4-bit numbers requires a much shorter time compared to adding two 16-bit numbers).

One way to avoid these problems is to opt for an asynchronous design approach and to eliminate all the clocks. Designing a purely asynchronous circuit is a nontrivial and potentially hazardous task. Ensuring a correct circuit operation that avoids all potential race conditions under any operation condition and input sequence requires a careful timing analysis of the network. In fact, the logical ordering of the events is dictated by the structure of the transistor network and the relative delays of the signals. Enforcing timing
constraints by manipulating the logic structure and the lengths of the signal paths requires an extensive use of CAD tools, and is only recommended when strictly necessary.

A more reliable and robust technique is the self-timed approach, which presents a local solution to the timing problem [Seitz80]. Figure 10.32 uses a pipelined datapath to illustrate how this can be accomplished. This approach assumes that each combinational function has a means of indicating that it has completed a computation for a particular piece of data. The computation of a logic block is initiated by asserting a Start signal. The combinational logic block computes on the input data and in a data-dependent fashion (taking the physical constraints into account) generates a Done flag once the computation is finished. Additionally, the operators must signal each other that they are either ready to receive a next input word or that they have a legal data word at their outputs that is ready for consumption. This signaling ensures the logical ordering of the events and can be achieved with the aid of an extra Acknowledge) and Request) signal. In the case of the pipelined datapath, the scenario could proceed as follows.

1. An input word arrives, and a Request) to the block F1 is raised. If F1 is inactive at that time, it transfers the data and acknowledges this fact to the input buffer, which can go ahead and fetch the next word.

2. F1 is enabled by raising the Start signal. After a certain amount of time, dependent upon the data values, the Done signal goes high indicating the completion of the computation.

3. A Request) is issued to the F2 module. If this function is free, an Acknowledge) is raised, the output value is transferred, and F1 can go ahead with its next computation.

The self-timed approach effectively separates the physical and logical ordering functions implied in circuit timing. The completion signal Done ensures that the physical timing constraints are met and that the circuit is in steady state before accepting a new input. The logical ordering of the operations is ensured by the acknowledge-request scheme, often called a handshaking protocol. Both interested parties synchronize with each other by mutual agreement or, if you want, by shaking hands. The ordering protocol described above and implemented in the module HS is only one of many that are possible. The choice of protocol is important, since it has a profound effect on the circuit performance and robustness.
When compared to the synchronous approach, self-timed circuits display some alluring properties.

- In contrast to the global centralized approach of the synchronous methodology, timing signals are generated \textit{locally}. This avoids all problems and overheads associated with distributing high-speed clocks.

- Separating the physical and logical ordering mechanisms results in a potential increase in performance. In synchronous systems, the period of the clock has to be stretched to accommodate the slowest path over all possible input sequences. In self-timed systems, a completed data-word does not have to wait for the arrival of the next clock edge to proceed to the subsequent processing stages. Since circuit delays are often dependent upon the actual data value, a self-timed circuit proceeds at the \textit{average speed} of the hardware in contrast to the \textit{worst-case} model of synchronous logic. For a ripple carry adder, the average length of carry-propagation is $O(\log N)$. This is a fact that can be exploited in self-timed circuits, while in a synchronous methodology, a worst case performance that varies linearly with the number of bits $O(N)$ must be assumed.

- The automatic shut-down of blocks that are not in use can result in power savings. Additionally, the power consumption overhead of generating and distributing high-speed clocks can be partially avoided. As discussed earlier, this overhead can be substantial. It is also possible to reduce the average clock load in synchronous processors through the use of gated clocks in which portions of the circuits that don’t compute in a cycle are shut off.

- Self-timed circuits are by nature robust to variations in manufacturing and operating conditions such as temperature. Synchronous systems are limited by their performance at the extremes of the operating conditions. The performance of a self-timed system is determined by the actual operating conditions.

Unfortunately, these nice properties are not for free; they come at the expense of a substantial circuit-level overhead, which is caused by the need to generate completion signals and the need for handshaking logic that acts as a local traffic agent to order the circuit events (see block HS in Figure 10.32). Both of these topics are treated in more detail in subsequent sections.

### 10.4.2 Completion-Signal Generation

A necessary component of self-timed logic is the circuitry to indicate when a particular piece of circuitry has completed its operation for the current piece of data. There are two common and reliable ways to generate the completion signal.

#### Dual-Rail Coding

One common approach to completion signal generation is the use of dual rail coding. It actually requires the introduction of redundancy in the data representation to signal that a particular bit is either in a transition or steady-state mode. Consider the redundant data
model presented in Table 10.1. Two bits ($B_0$ and $B_1$) are used to represent a single data bit $B$. For the data to be valid or the computation to be completed, the circuit must be in a legal 0 ($B_0 = 0$, $B_1 = 1$) or 1 ($B_0 = 1$, $B_1 = 0$) state. The ($B_0 = 0$, $B_1 = 0$) condition signals that the data is non-valid and the circuit is in either a reset or transition mode. The ($B_0 = 1$, $B_1 = 1$) state is illegal and should never occur in an actual circuit.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$B_0$</th>
<th>$B_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>in transition (or reset)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>illegal</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A circuit that actually implements such a redundant representation is shown in Figure 10.33, which is a dynamic version of the DCVSL logic style where the clock is replaced by the Start signal [Heller84]. DCVSL uses a redundant data representation by nature of its differential dual-rail structure. When the Start signal is low, the circuit is precharged by the PMOS transistors, and the output ($B_0$, $B_1$) goes in the Reset-Transition state (0, 0). When the Start signal goes high, signaling the initiation of a computation, the NMOS pull-down network evaluates, and one of the precharged nodes is lowered. Either $B_0$ or $B_1$—but never both—goes high, which raises Done and signals the completion of the computation.

DCVSL is more expensive in terms of area than a non-redundant circuit due to its dual nature. The completion generation is performed in series with the logic evaluation, and its delay adds directly to the total delay of the logic block. The completion signals of all the individual bits must be combined to generate the completion for an $N$-bit data word. Completion generation thus comes at the expense of both area and speed. The benefits of the dynamic timing generation often justify this overhead. Redundant signal presentations other than the one presented in Table 10.1 can also be envisioned. One essential element is the presence of a transition state denoting that the circuit is in evaluation mode and the output data is not valid.
An efficient implementation of a self-timed adder circuit is shown in Figure 10.34 [Abous93]. A Manchester-carry scheme is used to boost the circuit performance. The proposed approach is based on the observation that the circuit delay of an adder is dominated by the carry-propagation path. It is consequentially sufficient to use the differential signaling in the carry path only (Figure 10.34a). The completion signal is efficiently derived by combining the carry signals of the different stages (Figure 10.34b). This safely assumes that the sum generation, which depends upon the arrival of the carry signal, is faster than the completion generation. The benefit of this approach is that the completion generation starts earlier and proceeds in parallel with sum generation, which reduces the critical timing path. All other signals such as \textit{P}ropagate, \textit{G}enerate, \textit{K}ill, and \textit{S}um do not require completion generation and can be implemented in single-ended logic. As shown in the circuit schematics, the differential carry paths are virtually identical. The only difference is that the \textit{G}enerate signal is replaced by a \textit{K}ill. A simple logic analysis demonstrates that this indeed results in an inverted carry signal and, hence, a differential signaling.

![Manchester-carry scheme with differential signal representation.](image)

**Figure 10.34** Manchester-carry scheme with differential signal representation.

**Replica Delay**

While the dual-rail coding above allows tracking of the signal statistics, it comes at the cost of power dissipation. Every single gate must transition for every new input vector, regardless of the value of the data vector. An attempt to reduce the overhead of completion detection is to use a critical-path replica configured as a delay element, as shown in Figure
To start a computation, the Start signal is raised and the computation of the logic network is initiated. At the same time, the start signal is fed into the replica delay line, which tracks the critical path of the logic network. It is important that the replica is structured such that no glitching transitions occur. When the output of the delay line makes a transition, it indicates that the logic is complete—as the delay line mimics the critical path. In general, it is important to add extra padding in the delay line to compensate for possible random process variations. The advantage of this approach is that the logic can be implemented using a standard non-redundant circuit style such as complementary CMOS. Also, if multiple logic units are computing in parallel, it is possible to amortize the overhead of the delay line over multiple blocks. Note that this approach generates the completion signal after a time equal to the worst case delay through the network. As a result, it does not exploit the statistical properties of the incoming data. However, it can track the local effects of process variations and environmental variations (e.g., temperature or power supply variations). This approach is widely used to generate the internal timing of semiconductor memories where self-timing is a commonly used technique.

Example 10.5 An alternate completion detection circuit using current sensing

Ideally, logic should be implemented using non-redundant CMOS (e.g., static CMOS) and the completion signal circuitry should track data dependencies. Figure 10.36 shown an example technique that attempts to realize the above [REF]. A current sensor is inserted in series with the combinational logic, and monitors the current flowing through the logic. The current sensor outputs a low value when no current flows through the logic (this is, the logic is idle), and is high when the combinational logic is switching performing computation. This signal effectively determines when the logic has completed its cycle. Note that this approach tracks data-dependent computation times — if only the lower order bits of an adder switch, current will stop flowing once the lower order bits switch to the final value.

If the input data vector does not change from one cycle to the next, no current is drawn from the supply for static CMOS logic. In this case, a delay element that tracks the minimum delay through the logic and current sensor is used for signal completion. The outputs of the current sensor and minimum delay element are then combined. This approach is interesting, but requires careful analog design. Ensuring reliability while keeping the overhead circuitry small is the main challenge. These concerns have kept the applicability of the approach very limited, notwithstanding its great potential.
10.4.3 Self-Timed Signaling

Besides the generation of the completion signals, a self-timed approach also requires a handshaking protocol to logically order the circuit events avoiding races and hazards. The functionality of the signaling (or handshaking) logic is illustrated by the example of Figure 10.37, which shows a sender module transmitting data to a receiver ([Sutherland89]).

![Sender-receiver configuration](image1)

![Timing diagram](image2)

The sender places the data value on the data bus and produces an event on the Req control signal by changing the polarity of the signal. In some cases the request event is a rising transition; at other times it is a falling one—the protocol described here does not distinguish between them. Upon receiving the request, the receiver accepts the data when possible and produces an event on the Ack signal to indicate that the data has been accepted. If the receiver is busy or its input buffer is full, no Ack event is generated, and the transmitter is stalled until the receiver becomes available by, for instance, freeing space in the input buffer. Once the Ack event is produced, the transmitter goes ahead and produces the next data word. The four events, data change, request, data acceptance,
and acknowledge, proceed in a cyclic order. Successive cycles may take different amounts of time depending upon the time it takes to produce or consume the data.

This protocol is called two-phase, since only two phases of operation can be distinguished for each data transmission—the active cycle of the sender and the active cycle of the receiver. Both phases are terminated by certain events. The Req event terminates the active cycle of the sender, while the receiver’s cycle is completed by the Ack event. The sender is free to change the data during its active cycle. Once the Req event is generated, it has to keep the data constant as long as the receiver is active. The receiver can only accept data during its active cycle.

The correct operation of the sender-receiver system requires a strict ordering of the signaling events, as indicated by the arrows in Figure 10.37. Imposing this order is the task of the handshaking logic which, in a sense, performs logic manipulations on events. An essential component of virtually any handshaking module is the Muller C-element. This gate, whose schematic symbol and truth table are given in Figure 10.38, performs an AND-operation on events. The output of the C-element is a copy of its inputs when both inputs are identical. When the inputs differ, the output retains its previous value. Phrased in a different way—events must occur at both inputs of a Muller C-element for its output to change state and to create an output event. As long as this does not happen, the output remains unchanged and no output event is generated. The implementation of a C-element is centered around a flip-flop, which should be of no surprise, given the similarities in their truth tables. Figure 10.39 displays two potential circuit realizations—a static and a dynamic one respectively.

Figure 10.40 shows how to use this component to enforce the two-phase handshaking protocol for the example of the sender-receiver. Assume that Req, Ack, and Data Ready are initially 0. When the sender wants to transmit the next word, the Data Ready signal is set to 1, which triggers the C-element because both its inputs are at 1. Req goes
high—this is commonly denoted as \( \text{Req}^{\uparrow} \). The sender now resides in the wait mode, and control is passed to the receiver. The C-element is blocked, and no new data is sent to the data bus (\( \text{Req} \) stays high) as long as the transmitted data is not processed by the receiver. Once this happens, the \( \text{Data accepted} \) signal is raised. This can be the result of many different actions, possibly involving other C-elements communicating with subsequent blocks. An \( \text{Ack}^{\uparrow} \) ensues, which unblocks the C-element and passes the control back to the sender. A \( \text{Data ready}^{\downarrow} \) event, which might already have happened before \( \text{Ack}^{\uparrow} \), produces a \( \text{Req}^{\downarrow} \), and the cycle is repeated.

**Problem 10.1 Two-Phase Self-Timed FIFO**

Figure 10.41 shows a two-phase, self-timed implementation of a FIFO (first-in first-out) buffer with three registers. Assuming that the registers accept a data word on both positive- and negative-going transitions of the \( \text{En} \) signals and that the \( \text{Done} \) signal is simply a delayed version of \( \text{En} \), examine the operation of the FIFO by plotting the timing behavior of all signals of interest. How can you observe that the FIFO is completely empty (full)? (Hint: Determine the necessary conditions on the \( \text{Ack} \) and \( \text{Req} \) signals.)

The two-phase protocol has the advantage of being simple and fast. There is some bad news, however. This protocol requires the detection of transitions that may occur in either direction. Most logic devices in the MOS technology tend to be sensitive to levels or to transitions in one particular direction. Event-triggered logic, as required in the two-phase protocol, requires extra logic as well as state information in the registers and the computational elements. Since the transition direction is important, initializing all the Muller C-elements in the appropriate state is essential. If this is not done, the circuit might deadlock, which means that all elements are permanently blocked and nothing will ever
happen. A detailed study on how to implement event-triggered logic can be found in the text by Sutherland on micropipelines [Sutherland89].

The only alternative is to adopt a different signaling approach, called four-phase signaling, or return-to-zero (RTZ). This class of signaling requires that all controlling signals be brought back to their initial values before the next cycle can be initiated. Once again, this is illustrated with the example of the sender-receiver. The four-phase protocol for this example is shown in Figure 10.42.

![Figure 10.42 Four-phase handshaking protocol.](image)

The protocol presented is initially the same as the two-phase one. Both the Req and the Ack are initially in the zero-state, however. Once a new data word is put on the bus ①, the Req is raised (Req↑ or ②) and control is passed to the receiver. When ready, the receiver accepts the data and raises Ack (Ack↑ or ③). So far, nothing new. The protocol proceeds, now by bringing both Req (Req↓ or ④) and Ack (Ack↓ or ⑤) back to their initial state in sequence. Only when that state is reached is the sender allowed to put new data on the bus ①. This protocol is called four-phase because four distinct time-zones can be recognized per cycle: two for the sender; two for the receiver. The first two phases are identical to the two-phase protocol; the last two are devoted to resetting of the state. An implementation of the protocol, based on Muller C-elements, is shown in Figure 10.43. It is interesting to notice that the four-phase protocol requires two C-elements in series (since four states must be represented). The Data ready and Data accepted signals must be pulses instead of single transitions.

![Figure 10.43 Implementation of 4-phase handshake protocol using Muller C-elements.](image)
### Problem 10.2 Four-Phase Protocol

Derive the timing diagram for the signals shown in Figure 10.43. Assume that the Data Ready signal is a pulse and that the Data Accepted signal is a delayed version of Req.

The four-phase protocol has the disadvantage of being more complex and slower, since two events on Req and Ack are needed per transmission. On the other hand, it has the advantage of being robust. The logic in the sender and receiver modules does not have to deal with transitions, which can go either way, but only has to consider rising (or falling) transition events or signal levels. This is readily accomplished with traditional logic circuits. For this reason, four-phase handshakes are the preferred implementation approach for most of the current self-timed circuits. The two-phase protocol is mostly selected when the sender and receiver are far apart and the delays on the control wires (Ack, Req) are substantial.

### Example 10.6 The Pipelined Datapath—Revisited

We have introduced both the signaling conventions and the concepts of the completion-signal generation. Now it is time to bring them all together. We do this with the example of the pipelined data path, which was presented earlier. A view of the self-timed data path, including the timing control, is offered in Figure 10.44. The logic functions F1 and F2 are implemented using dual-rail, differential logic.

To understand the operation of this circuit, assume that all Req and Ack signals, including the internal ones, are set to 0, what means there is no activity in the data path. All Start signals are low so that all logic circuits are in precharge condition. An input request (Req↑) triggers the first C-element. The enable signal En of R1 is raised, effectively latching the input data into the register, assuming a positive edge-triggered or a level-sensitive implementation. Ack↑ acknowledges the acceptance of the data. The second C-element is triggered as well, since Ackint is low. This raises the Start signal and starts the evaluation of F1. At its completion, the output data is placed on the bus, and a request is initiated to the second stage (Reqint↑), which acknowledges its acceptance by raising Ackint↑.

At this point, stage 1 is still blocked for further computations. However, the input buffer can respond to the Ack↑ event by resetting Req to its zero state (Req↓). In turn, this lowers En and Ack↓. Upon receipt of Ackint↑, Start goes low, the pre-charge phase starts, and...
$F1$ is ready for new data. Note that this sequence corresponds to the four-phase handshake mechanism described earlier. The dependencies among the events are presented in a more pictorial fashion in the state transition diagram (STG) shown in Figure 10.45. These STGs can become very complex. Computer tools are often used to derive STGs that ensure proper operation and optimize the performance.

10.4.4 Practical Examples of Self-Timed Logic

As described in the previous section, self-timed circuits can provide significant performance advantage. Unfortunately, the overhead circuits preclude widespread application in general purpose digital computing. However, several applications exist that exploit the key concepts of self-timed circuits. A few examples that illustrate the use of self-timed concepts for either power savings or performance enhancement are presented below.

**Glitch Reduction Using Self-timing**

A major source of unnecessary switched capacitance in a large datapath such as bit-sliced adders and multipliers, is due to spurious transitions caused by glitch propagation. Imbalances in a logic network cause inputs of a logic gate or block to arrive at different times, resulting in glitching transitions. In large combinational blocks such as multipliers, transitions happens in waves as the primary input changes ripple through the logic. If a logic block can be enabled after all of the inputs settle, then the number of glitching transitions can be reduced. One approach for minimizing spurious transitions is the use of a self-timed gating approach that involves partitioning each computational logic block into smaller blocks and distinct phases. Tri-state buffers are inserted between each of these phases to prevent glitches from propagating further in the datapath (Figure 10.46). Assuming an arbitrary logic network in Figure 10.46, the outputs of logic block 1 will not be synchronized. When the tri-state buffers at the output of logic block 1 are enabled, the the computation of logic block 2 is allowed to proceed. To reduce glitching transitions, the tri-state buffer should be enabled only when the outputs of logic block 1 are ensured to be stable and valid. The control of the tri-state buffer can be performed through the use of a self-timed enable signal which is generated by passing the system clock through a delay chain.

![State transition diagram for pipeline stage 1. The nodes represent the signaling events, while the arrows express dependencies. Arrows in dashed lines express actions in either the preceding or following stage.](image)
that models the critical path of the processor. The delay chain is then tapped at the points corresponding to the locations of the buffers, and the resulting signals are distributed throughout the chip. This technique succeeds in reducing the switched capacitance combina
tional logic blocks such as multipliers, even including the overhead of the delay chain, gating signal distribution and buffers [Goodman98].

Post-Charge Logic

An interesting form of self-timed logic is self-resetting CMOS. This structure uses a dif
derent control structure than the conventional self-timed pipeline described earlier. The idea is that instead of checking if all of the logic blocks have completed their operation before transitioning to the reset stage, the idea is to precharge a logic block as soon as it completes it operation. Since the precharge happens after operation instead of before evalua
tion, it is often termed *post-charge logic*. A block diagram of post-charge logic is shown in Figure 10.47 [Williams00]. As seen from this block diagram, the precharging of L1

![Figure 10.46 Application of self-timing for glitch reduction.](image)

![Figure 10.47 Self-resetting logic.](image)

happen when the successor stage has finished evaluating. It is possible to precharge a block based on the completion of its own output, but care must be taken to ensure that the following stage has properly evaluated based on the output before it is precharged.
It should be noted that unlike other logic styles, the signals are represented as pulses, and are valid only for a limited duration. The pulse width must be shorter than the reset delay or else, there would be a contention between the precharge device and the evaluate switches. While this logic style offers potential speed advantages, special care must be taken to ensure correct timing. Also, circuitry that converts level signals to pulses are required. An example of self-resetting logic is shown in Figure 10.48 [Bernstein98]. Assume that all inputs are low, and \( \text{int} \) is initially precharged. If \( A \) goes high, \( \text{int} \) will fall, causing \( \text{out} \) to go high. This causes, the gate to precharge. When the PMOS precharge device is active, the inputs must be in a reset state to avoid contention.

**Clock-Delayed Domino**

One interesting application of self-timed circuits using the delay-matching concept is Clock-Delayed Domino. This is a style of dynamic logic, where there is no global clock signal. Instead, the clock for one stage is derived from the previous stage. A block diagram of this is shown in Figure 10.49. The two inverter delays along the clock path emulate the worst case delay through the logic path (i.e., the Pulldown Network). Sometimes, there is a transmission gate inserted between the two inverters to accomplish this (the transmission gate is on with the gate terminal of NMOS tied to \( V_{DD} \) and the PMOS to \( GND \)). Clock-delayed Domino was used in the IBM’s 1GHz Microprocessor and is used widely in high speed Domino logic. There are several advantages of using such a self-clocked timing abstraction. First, CD domino can provide both inverting and non-inverting function. This alleviates a major limitation of conventional Domino that is only capable of non-inverting logic. The inverter after the pulldown network is not essential as the clock arrives to the next stage only after the current stage has evaluated. Also, notice that it is possible to eliminate the “foot-switch” in the later stages, as the clock-evaluation edge arrives only when the input is stable. This provides significant speed up of the logic critical path. A careful analysis of the timing shows that the short circuit power can be eliminated.
10.5 Synchronizers and Arbiters*

10.5.1 Synchronizers—Concept and Implementation

Even though a complete system may be designed in a synchronous fashion, it must still communicate with the outside world, which is generally asynchronous. An asynchronous input can change value at any time related to the clock edges of the synchronous system, as is illustrated in Figure 10.50.

![Figure 10.50 Asynchronous-synchronous interface](image)

Consider a typical personal computer. All operations within the system are strictly orchestrated by a central clock that provides a time reference. This reference determines what happens within the computer system at any point in time. This synchronous computer has to communicate with a human through the mouse or the keyboard, who has no knowledge of this time reference and might decide to press a keyboard key at any point in time. The way a synchronous system deals with such an asynchronous signal is to sample or poll it at regular intervals and to check its value. If the sampling rate is high enough, no transitions will be missed—this is known as the Nyquist criterion in the communication community. However, it might happen that the signal is polled in the middle of a transition. The resulting value is neither low or high but undefined. At that point, it is not clear if the key was pressed or not. Feeding the undefined signal into the computer could be the source of all types of trouble, especially when it is routed to different functions or gates that might interpret it differently. For instance, one function might decide that the key is pushed and start a certain action, while another function might lean the other way and issue a competing command. This results in a conflict and a potential crash. Therefore, the undefined state must be resolved in one way or another before it is interpreted further. It does not really matter what decision is made, as long as a unique result is available. For instance, it is either decided that the key is not yet pressed, which will be corrected in the next poll of the keyboard, or it is concluded that the key is already pressed.

Thus, an asynchronous signal must be resolved to be either in the high or low state before it is fed into the synchronous environment. A circuit that implements such a decision-making function is called a synchronizer. Now comes the bad news—**building a perfect synchronizer that always delivers a legal answer is impossible!** [Chaney73, Glasser85] A synchronizer needs some time to come to a decision, and in certain cases this time might be arbitrarily long. An asynchronous/synchronous interface is thus always prone to errors called synchronization failures. The designer’s task is to ensure that the probability of such a failure is small enough that it is not likely to disturb the normal system behavior. Typically, this probability can be reduced in an exponential fashion by waiting longer before
making a decision. This is not too troublesome in the keyboard example, but in general, waiting affects system performance and should therefore be avoided to a maximal extent.

To illustrate why waiting helps reduce the failure rate of a synchronizer, consider a synchronizer as shown in Figure 10.51. This circuit is a latch that is transparent during the low phase of the clock and samples the input on the rising edge of the clock $CLK$. However, since the sampled signal is not synchronized to the clock signal, there is a finite probability that the set-up time or hold time of the latch is violated (the probability is a strong function of the transition frequencies of the input and the clock). As a result, one the clock goes high, there is a the chance that the output of the latch resides somewhere in the undefined transition zone. The sampled signal eventually evolves into a legal 0 or 1 even in the latter case, as the latch has only two stable states.

**Example 10.7 Flip-Flop Trajectories**

Figure 10.52 shows the simulated trajectories of the output nodes of a cross-coupled static CMOS inverter pair for an initial state close to the metastable point. The inverters are composed of minimum-size devices.

![Figure 10.52 Simulated trajectory for a simple flip-flop synchronizer.](image)

If the input is sampled such that cross-coupled inverter starts at the metastable point, the voltage will remain at the metastable state forever in the absense of noise. If the data is sampled with a small offset (positive or negative), the differential voltage will evolve in an exponential form, with a time constant that is dependent on the strength of the transistor as well as the parasitic capacitances. The time it takes to reach the acceptable signal zones depends upon the initial distance of the sampled signal from the metastable point.

In order to determine the required waiting period, let us build a mathematical model of the behavior of the bistable element and use the results to determine the probability of synchronization failure as a function of the waiting period. This model is used to compute the range of values for $v(0)$ that still cause an error, or a voltage in the undefined range, after a waiting period $T$. A signal is called undefined if its value is situated between $V_{IH}$ and $V_{IL}$. 

![Figure 10.51 A simple synchronizer.](image)
Section 10.5 Synchronizers and Arbiters

Eq. (10.8) conveys an important message: The range of input voltages that cause a synchronization error decreases exponentially with the waiting period $T$. Increasing the waiting period from $2\tau$ to $4\tau$ decreases the interval and the chances of an error by a factor of 7.4.

Some information about the asynchronous signal is required in order to compute the probability of an error. Assume that $V_{in}$ is a periodical waveform with an average period $T_{signal}$ between transitions and with identical rise and fall times $t_r$. Assume also that the slopes of the waveform in the undefined region can be approximated by a linear function (Figure 10.53). Using this model, we can estimate the probability $P_{init}$ that $v(0)$, the value of $V_{in}$ at the sampling time, resides in the undefined region.

\[ P_{init} = \frac{(V_{IH} - V_{IL}) t_r}{V_{swing} T_{signal}} \]  

The chances for a synchronization error to occur depend upon the frequency of the synchronizing clock $\phi$. The more sampling events, the higher the chance of running into an error. This means that the average number of synchronization errors per second $N_{sync}(0)$ equals Eq. (10.10) if no synchronizer is used.

\[ N_{sync}(0) = \frac{P_{init}}{T_\phi} \]  

where $T_\phi$ is the sampling period.

From Eq. (10.8) we learned that waiting a time $T$ before observing the output reduces exponentially the probability that the signal is still undefined.

\[ N_{sync}(T) = \frac{P_{init} e^{-T/\tau}}{T_\phi} = \frac{(V_{IH} - V_{IL}) e^{-T/\tau}}{V_{swing} T_{signal} T_\phi} \]

The robustness of an asynchronous-synchronous interface is determined by the following parameters: signal switching rate and rise time, sampling frequency, and waiting time $T$.

**Example 10.8 Synchronizers and Mean Time-to-Failure**

Consider the following design example. $T_\phi = 5$ nsec, which corresponds to a 200 Mhz clock), $T = T_\phi = 5$ nsec, $T_{signal} = 50$ nsec, $t_r = 0.5$ nsec, and $\tau = 150$ psec (as obtained in Example 10.7).
From the VTC of a typical CMOS inverter, it can be derived that $V_{IH} - V_{IL}$ approximately equals 0.5 V for a voltage swing of 2.5 V. Evaluation of Eq. (10.11) yields an error probability of $1.38 \times 10^{-9}$ errors/sec. The inverse of $N_{sync}$ is called the mean time-to-failure, or the MTF, and equals $7 \times 10^8$ sec, or 23 years. If no synchronizer was used, the MTF would only have been 2.5 μsec.

**Design Consideration**

When designing a synchronous/asynchronous interface, we must keep in mind the following observations:

- The acceptable failure rate of a system depends upon many economic and social factors and is a strong function of its application area.

- The exponential relation in Eq. (10.11) makes the failure rate extremely sensitive to the value of $\tau$. Defining a precise value of $\tau$ is not easy in the first place. $\tau$ varies from chip to chip and is a function of temperature as well. The probability of an error occurring can thus fluctuate over large ranges even for the same design. A worst-case design scenario is definitely advocated here. If the worst-case failure rate exceeds a certain criterion, it can be reduced by increasing the value of $T$. A problem occurs when $T$ exceeds the sampling period $T_{\phi}$. This can be avoided by cascading (or pipelining) a number of synchronizers, as shown in Figure 10.54. Each of those synchronizers has a waiting period equal to $T_{\phi}$. Notice that this arrangement requires the $\phi$-pulse to be short enough to avoid race conditions. The global waiting period equals the sum of the $T_s$ of all the individual synchronizers. The increase in MTF comes at the expense of an increased latency.

**Figure 10.54** Cascading synchronizers reduces the main time-to-failure.

- Synchronization errors are very hard to trace due to their probabilistic nature. Making the mean time-to-failure very large does not preclude errors. The number of synchronizers in a system should therefore be severely restricted. A maximum of one or two per system is advocated.

### 10.5.2 Arbiters

Finally, a sibling of the synchronizer called the *arbiter*, interlock element, or mutual-exclusion circuit, should be mentioned. An arbiter is an element that decides which of two events has occurred first. Such components for instance allow multiple processors to access a single resource, such as a large shared memory. A synchronizer is actually a
special case of an arbiter, since it determines if a signal transition happened before or after a clock event. A synchronizer is thus an arbiter with one of its inputs tied to the clock.

An example of a mutual-exclusion circuit is shown in Figure 10.55. It operates on two input-request signals, that operate on a four-phase signaling protocol; that is, the Req(uest) signal has to go back to the reset state before a new Req(uest) can be issued. The output consists of two Ack(nowledge) signals that should be mutually exclusive. While Requests may occur concurrently, only one of the Acknowledges is allowed to go high. The operation is most easily visualized starting with both inputs low—neither device issuing a request—nodes A and B high, and both Acknowledges low. An event on one of the inputs (e.g., Req1↑) causes the flip-flop to switch, node A goes low, and Ack1↑. Concurrent events on both inputs force the flip-flop into the metastable state, and the signals A and B might be undefined for a while. The cross-coupled output structure keeps the output values low until one of the NAND outputs differs from the other by more than a threshold value \( V_T \). This approach eliminates glitches at the output.

![Mutual-exclusion element (or arbiter).](image)

### Figure 10.55 Mutual-exclusion element (or arbiter).

10.6 Clock Synthesis and Synchronization Using a Phase-Locked Loop

There are numerous digital applications that require the on-chip generation of a periodic signal. Synchronous circuits need a global periodic clock reference to drive sequential elements. Current microprocessors and high performance digital circuits require clock frequencies in the gigahertz range. Crystal oscillators generate accurate, low-jitter clocks with a frequency range from 10’s of Megahertz to approximately 200MHz. To generate a higher frequency required by digital circuits, a phase-locked loop (PLL) structure is typically used. A PLL takes an external low-frequency reference crystal frequency signal and multiplies its frequency by a rational number \( N \) (see the left side of Figure 10.56).
PLLs are also used to perform synchronization of communication between chips. Typically as shown in Figure 10.56, a reference clock is sent along with the parallel data being communicated (in this example only the transmit path from chip 1 to chip 2 is shown). Since chip-to-chip communication occurs at a lower rate than the on-chip clock rate, the reference clock is a divided but in-phase version of the system clock. The reference clock synchronizes all input flip-flops on chip 2; this can present a significant clock load for wide data busses. Introducing clock buffers to deal with this problem unfortunately introduces skew between the data and sample clock. A PLL, using feedback, can be align (i.e., de-skew) the output of the clock buffer with respect to the data. In addition, for the configuration shown in Figure 10.56, the PLL can multiply the frequency of the incoming reference clock, allowing the reference clock to be a fraction of the data rate.

![ PLL Applications Diagram ]

**Figure 10.56** Applications of Phase Locked Loops (PLL).

### 10.6.1 Basic Concept

Periodic signals of known frequency can be described exactly by only one parameter, their phase. More accurately a set of two or more periodic signals of the same frequency can be well defined if we know one of them and its phase with respect to the other signals, as in Figure 10.57. Here $\phi_1$ and $\phi_2$ represent the phase of the two signals. The relative phase is defined as the difference between the two phases.

![ Relative and Absolute Phase Diagram ]

**Figure 10.57** Relative and absolute phase of two periodic signals
A PLL is a complex, nonlinear feedback circuit, and its basic operation is understood with the aid of Figure 10.58 [Jeong87]. The voltage-controlled oscillator (VCO) takes an analog control input and generates a clock signal of the desired frequency. In general, there is a non-linear relationship between the control voltage \( v_{\text{cont}} \) and the output frequency. To synthesize a system clock of a particular frequency, it necessary to set the control voltage to the appropriate value. This is function of the rest of the blocks (i.e., feedback loop) in the PLL. The feedback loop is critical to tracking process and environmental variations. The feedback also allows frequency multiplication.

The reference clock is typically generated off-chip from an accurate crystal reference. The reference clock is compared to a divided version of the system clock (i.e., the local clock). The local clock and reference clock are compared using a phase detector that compares the phase difference between the signals and produces an \( Up \) or \( Down \) signal when the local clock lags or leads the reference signal. It detects which of the two input signals arrives earlier and produces an appropriate output signal. Next, the \( Up \) and \( Down \) signals are fed into a charge pump, which translates the digital encoded control information into an analog voltage [Gardner80]. An \( Up \) signal increases the value of the control voltage and speeds up the VCO, which causes the local signal to catch up with the reference clock. A \( Down \) signal, on the other hand, slows down the oscillator and eliminates the phase lead of the local clock.

Passing the output of the charge pump directly into the VCO creates a jittery clock signal. The edge of the local clock jumps back and forth instantaneously and oscillates around the targeted position. As discussed earlier, clock jitter, is highly undesirable, since it reduces the time available for logic computation and therefore should be kept within a given percentage of the clock period. This is partially accomplished by the introduction of the loop filter. This low-pass filter removes the high-frequency components from the VCO control voltage and smooths out its response, which results in a reduction of the jitter. Note that the PLL structure is a feedback structure and the addition of extra phase shifts, as is done by a high-order filter, may result in instability. Important properties of a PLL are its lock range—the range of input frequencies over which the loop can maintain functionality; the lock time—the time it takes for the PLL to lock onto a given input signal; and the jitter. When in lock, the system clock is \( N \)-times the reference clock frequency.

A PLL is an analog circuit and is inherently sensitive to noise and interference. This is especially true for the loop filter and VCO, where induced noise has a direct effect on the resulting clock jitter. A major source of interference is the noise coupling through the supply
Timings and the substrate. This is particularly a concern in digital environments, where noise is introduced due to a variety of sources. Analog circuits with a high supply rejection, such as differential VCOs, are therefore desirable [Kim90]. In summary, integrating a highly sensitive component into a hostile digital environment is nontrivial and requires expert analog design. Below a more detailed description of various components of a PLL is given.

10.6.2 Building Blocks of a PLL

Voltage Controlled Oscillator (VCO)

A VCO generates a periodic signal, whose frequency is a linear function of the input control voltage $v_{\text{cont}}$. In other words, a VCO is characterized by $\omega = \omega_0 + K_{\text{vco}} \cdot v_{\text{cont}}$. As the phase is the time integral of the frequency, the output phase of the VCO block is given by,

$$\phi_{\text{out}} = \omega_0 t + K_{\text{vco}} \int_{-\infty}^{t} v_{\text{cont}} dt$$

(10.12)

where $K_{\text{vco}}$ is the gain of the VCO given in rad/s/V, and $\omega_0$ is a fixed frequency offset. $v_{\text{cont}}$ controls a frequency centered around $\omega_0$. The output signal has the form

$$x(t) = A \cdot \cos \left( \omega_0 t + K_{\text{vco}} \int_{-\infty}^{t} v_{\text{cont}} dt \right)$$

(10.13)

Phase Detectors

The phase detector determines the relative phase difference between two incoming signals and outputs a signal that is proportional to this phase difference. This output signal is then used to adjust the output of the VCO and thus align the two inputs via a feedback network. One of the inputs to the phase detector is a reference clock that is typically generated off-chip while the other clock input is a divided version of the VCO. Two basic types of phase detectors are commonly used. These include the XOR gate and the phase frequency detector (PFD).

XOR Phase Detector. The XOR phase detector is the simplest phase detector. The XOR is useful as a phase detector since the time when the two inputs are different (or same) represents the relative phase. Figure 10.59 shows the XOR of two waveforms. The output of
the XOR is low pass filtered and acts as a control voltage to the VCO. The output (low pass filtered) as a function of the phase error is also shown.

![Diagram of XOR as a phase detector]

Figure 10.59  The XOR as a phase detector.

For this detector any deviation in a positive or negative direction from the perfect in-phase condition (i.e., phase error of zero) produces the same change in duty factor resulting in the same average voltage. Thus the linear phase range is only 180 degrees. Using such a phase detector, a PLL will lock to a quadrature phase relationship (i.e., 1/4 cycle offset) between the two inputs. A drawback of the XOR phase detector is that it may lock to a multiple of the clock frequency. If the local clock is a multiple of the reference clock frequency, the output of the phase detector will still be a square wave of 50% duty cycle, albeit at a different frequency. The filtered version of this signal will be identical to that of the truly locked state and thus the VCO will operate at the nominal frequency.

**Phase-Frequency Detector.** The phase-frequency detector (PFD) is the most commonly used form of phase detector, and it solves several of the shortcomings of the detectors discussed above. As the name implies, the output of the PFD is dependent both on the phase and frequency difference of the applied signals. Accordingly, it cannot lock to an incorrect
multiple of the frequency. The PFD takes two clock inputs and produces two outputs, UP and DOWN as shown in Figure 10.60.

![Figure 10.60 Phase-Frequency Detector. (a) Schematic (b) State Transition Diagram (c) Timing.](image)

The PFD is a state machine with 3 states. Assume that both UP and DN outputs are initially low. When input A leads B, the UP output is asserted on the rising edge of input A. The UP signal remain in this state until a low-to-high transition occurs on input B. At that time, the DN output is asserted, causing both flip-flops to reset through the asynchronous reset signal. Notice that a short pulse proportional to the phase error is generated on the DN signal, and that there is a small pulse on the DN output, whose duration is equal to the delay through the AND gate and register reset delay. The pulse width of the UP pulse is equal to the phase error between the two signal. The roles are reversed for the case when input B lags A, and a pulse proportional to the phase error is generated on the DN output. If the loop is in lock, short pulses will be generated on the UP and DN outputs.

![Figure 10.61 Timing of the PFD measuring frequency error.](image)

The circuit also acts as a frequency detector, providing a measure of the frequency error (Figure 10.61). For the case when A is at a higher frequency than B, the PFD generates a lot more UP pulses (with the average proportional to the frequency difference),
Section 10.6  Clock Synthesis and Synchronization Using a Phase-Locked Loop

while the DN pulses average close to zero. The exactly the opposite is true for the case when B has a frequency larger than A — many more pulses are generated on the DN output than the UP output.

The phase characteristics of the phase detector is shown in Figure 10.62. Notice that the linear range has been expanded to \(4\pi\).

**Figure 10.62** Phase detector characteristic.

**Charge Pump**

The UP/DN pulses must be converted to an analog voltage that controls the VCO. One possible implementation is shown in Figure 10.63. A pulse on the UP signal adds a charge packet proportional to the size of the UP pulse, and a pulse on the DN signal removes a charge packet proportional to the DN pulse. If the width of the UP pulse is large than the DN pulse, then there is a net increase in the control voltage. This effectively increases the frequency of the VCO.

**Simulation**

The settling time of a PLL is the time it takes it reach steady state behavior. The period of the startup transient is strongly dependent on the bandwidth of the loop filter. Figure 10.64 shows a spice level simulation of a PLL (an ideal VCO is used to speed up the simulation) implemented in 0.25 \(\mu\)m CMOS. In this example a reference frequency of 100MHz is chosen and the PLL multiplies this frequency by 8 to 800MHz. The figure illustrates the transient response and settling process of the control voltage to the VCO. Once the control voltage reaches its final value, the output frequency (the clock to the digital system) settles to its final value. The simulation on the right shows the reference frequency, the output of the divider and the output frequency. As illustrated in this plot, the top graph show lock in which \(f_{out} = 8 \cdot f_{ref}\). The lower graph shows the waveforms during the locking process where the output is not in phase with the input and the frequencies are not related through the divide ratio.
Summary

In a short span of time, phased-locked loops have become an essential component of any high-performance digital design. Their design requires considerable skill, integrating analog circuitry into a hostile digital environment. Yet, experience has demonstrated that this combination is perfectly feasible, and leads to new and better solutions.

10.7 Future Directions

This section highlights some of the trends in high-performance and low-power timing optimization.

10.7.1 Distributed Clocking Using DLLs

A recent trend in high-performance clocking is the use of Delay-Locked Loop (DLL) structures, a small variation of the PLL structure. A schematic of a DLL is shown in Figure 10.65 [Maneatis00]. The key component of a DLL is a voltage-controlled delay line (VCDL). It consists of a cascade of adjustable delay elements (for instance, a current-starved inverter). The idea is to delay the output clock such that it perfectly lines up with the reference. Unlike a VCO, there is no clock generator. The reference frequency is fed into the input of the VCDL. Similar to a PLL structure, a phase-detector compares the reference frequency to the output of the delay line (F0), and generates an UP/DN error signal. Note that only a phase detection is required instead of a PFD. When in lock there is no
error between the two clocks. The function of the feedback is to adjust the delay through the VCDL such that the rising edge of the input reference clock ($f_{\text{REF}}$), and the output clock $f_O$ are aligned.

A qualitative sketch of the signals in the DLL is shown in Figure 10.66c. Initially, the DLL is out of lock. Since the first edge of the output arrives before the reference edge, an UP pulse of width equal to the error between the two signals. The role of the charge pump is to generate a charge packet proportional to the error, increasing the voltage of the VCDL control voltage. This causes the edge of the output signal to be delayed in the next cycle (this implementation of the VCDL assumes that a large voltage results in larger delay). After many cycles, the phase error is corrected, and the two signals are in lock.

Figure 10.67 shows the application of a DLL structure to clock distribution. The chip is partitioned into many small regions (or tiles). A global clock is distributed to each tile in a low-skew manner (e.g., this could be done through the package or using low skew on-chip routing schemes). For purpose of simplicity, the Figure shows a two-tile chip, but this is easily extended to many regions. Inside each tile, the global clock is buffered before driving the digital load. In front of each buffer is a VCDL. The goal of the clock network is to deliver a signal to the digital circuit with close-to-zero skew and jitter. Unfortunately, the static and dynamic variations of the buffers cause the phase error between the buffered clocks to be non-zero and time-varying. This is especially a concern since the delay through the buffer chain can equal multiple clock cycles. The feedback inside each tile adjusts the control voltage of VCDL, such that the buffered output is locked in phase to the global input clock. The feedback loop compensates for both static process variations and for slow dynamic variations (e.g., temperature). Such configurations have become common in high performance digital microprocessors.
10.7.2 Optical Clock Distribution

It is clear that there are some fundamental problems associated with electrical synchronization techniques for future high-performance multi-GHz systems. The performance of a digital is fundamentally limited by process and environmental variations. Even with aggressive active clock management schemes such as the use of DLLs and PLLs, the variations in power supply and clock load result in unacceptable clock uncertainty. As a result, there has been a major recent push towards the use of optics for system wide synchronization. An excellent review of the rationale and trade-offs in optical interconnects vs. electrical interconnect is given in [Miller00].

The potential advantage of optical technology for clock distribution is due to the fact that the delay is not sensitive to temperature and the clock edges don’t degrade over long distances (e.g., 10’s meters). In fact, it is possible to deliver an optical clock signal with 10-100ps of uncertainty for 10’s meters. Of course, the performance of an optical system is limited by the speed of light. Optical clocks can distribution on-chip via waveguides or freespace. Figure 10.67 shows the plot of an optical clock architecture using waveguides. The off-chip optical source is brought to the chip, distributed through waveguides, and converted through receiver circuitry to a local electrical clock distribution network. The chip is divided into small sections (or tiles) and each the global clock is distributed from the photon source (the most popular choice being a laser for its ease of use) through waveguides with splitters and bends to each of the sections. Notice that an H-treene is used in distributing the optical clock. At the end of the waveguide is a photodetector (which can be implemented using silicon or germanium). Once reaching the detector in each section, the global clock optical pulses are converted into current pulses. These current pulses have a very small magnitude (10’s µA) and they are fed into a transimpedance amplifier of the
Section 10.8 Perspective: Synchronous versus Asynchronous Design

The self-timed approach offers a potential solution to the growing clock-distribution problem. It translates the global clock signal into a number of local synchronization problems. Independence from physical timing constraints is achieved with the aid of completion signals. Handshaking logic is needed to ensure the logical ordering of the circuit events and to avoid race conditions. This requires adherence to a certain protocol, which normally consists of either two or four phases.

Despite all its advantages, self-timing has only been used in a number of isolated cases. Examples of self-timed circuits can be found in signal processing [Jacobs90], fast arithmetic units (such as self-timed dividers) [Williams87], simple microprocessors [Martin89] and memory (static RAM, FIFOs). In general, synchronous logic is both faster and simpler since the overhead of completion-signal generation and handshaking logic is avoided. The design a fool-proof network of handshaking units, that is robust with respect to races, live-lock, and dead-lock, is nontrivial and requires the availability of dedicated design-automation tools.

On the other hand, distributing a clock at high speed becomes exceedingly difficult. This was amply illustrated by the example of the 21164 Alpha microprocessor. Skew man-
agement requires extensive modeling and analysis, as well as careful design. It will not be easy to extend this methodology into the next generation of designs. This observation is already reflected in the fact that the routing network for the latest generation of massively parallel supercomputers is completely implemented using self-timing [Seitz92]. For self-timing to become a mainstream design technique however (if it ever will), further innovations in circuit and signaling techniques and design methodologies are needed. Other alternative timing approaches might emerge as well. Possible candidates are fully asynchronous designs or islands of synchronous units connected by an asynchronous network.

10.9 Summary

This chapter has explored the timing of sequential digital circuits.

- An in-depth analysis of the synchronous digital circuits and clocking approaches was presented. Clock skew and jitter has a major impact on the functionality and performance of a system. Important parameters are the clocking scheme used and the nature of the clock-generation and distribution network.

- Alternative timing approaches, such as self-timed design, are becoming attractive to deal with clock distribution problems. Self-timed design uses completion signals and handshaking logic to isolate physical timing constraints from event ordering.

- The connection of synchronous and asynchronous components introduces the risk of synchronization failure. The introduction of synchronizers helps to reduce that risk, but can never eliminate it.

- Phase-locked loops are becoming an important element of the digital-designer’s toolbox. They are used to generate high speed clock signals on a chip. The analog nature of the PLL makes its design a real challenge.

- Important trends for clock distribution include the use of delay-locked loops to actively adjust delays on a chip.

- The key message of this chapter is that synchronization and timing are among the most intriguing challenges facing the digital designer of the next decade.

10.10 To Probe Further

While system timing is an important topic, no congruent reference work is available in this area. One of the best discussions so far is the chapter by Chuck Seitz in [Mead80, Chapter 7]. Other in-depth overviews are given in [Bakoglu90, Chapter 8], [Johnson93, Chapter 11], [Hatamian88], and [Chandrakasan00]. A collection of papers on clock distribution networks is presented in [Friedman95]. Numerous other publications are available on this topic in the leading journals, some of which are mentioned below.
REFERENCES


[Bailey00] D. Bailey, “Clock Distribution,” in [Chandrakasan00].


[Bernstein00] K. Bernstein, “Basic Logic Families”, in [Chandrakasan00].


**EXERCISES AND DESIGN PROBLEM**

Please refer to the book web-page (http://bwrc.eecs.berkeley.edu/IcBook) for insightful and challenging exercises and design problems.