7.11 Timing methodologies

(Original section[1])

Throughout this appendix and in the rest of the text, we use an edge-triggered timing methodology. This timing methodology has an advantage in that it is simpler to explain and understand than a level-triggered methodology. In this section, we explain this timing methodology in a little more detail and also introduce level-sensitive clocking. We conclude this section by briefly discussing the issue of asynchronous signals and synchronizers, an important problem for digital designers.

The purpose of this section is to introduce the major concepts in clocking methodology. The section makes some important simplifying assumptions; if you are interested in understanding timing methodology in more detail, consult one of the references listed at the end of this appendix.

We use an edge-triggered timing methodology because it is simpler to explain and has fewer rules required for correctness. In particular, if we assume that all clocks arrive at the same time, we are guaranteed that a system with edge-triggered registers between blocks of combinational logic can operate correctly without races if we simply make the clock long enough. A *race* occurs when the contents of a state element depend on the relative speed of different logic elements. In an edge-triggered design, the clock cycle must be long enough to accommodate the path from one flip-flop through the combinational logic to another flip-flop where it must satisfy the setup-time requirement. The figure below shows this requirement for a system using rising edge-triggered flip-flops. In such a system the clock period (or cycle time) must be at least as large as

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}}$$

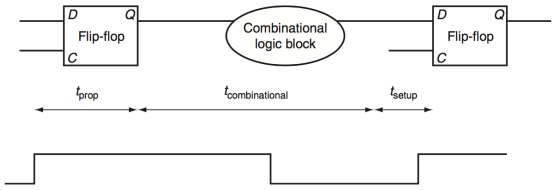
for the worst-case values of these three delays, which are defined as follows:

- t_{prop} is the time for a signal to propagate through a flip-flop; it is also sometimes called clock-to-Q.
- $t_{\text{combinational}}$ is the longest delay for any combinational logic (which by definition is surrounded by two flip-flops).
- t_{setup} is the time before the rising clock edge that the input to a flip-flop must be valid.

We make one simplifying assumption: the hold-time requirements are satisfied, which is almost never an issue with modern logic.

Figure 7.11.1: In an edge-triggered design, the clock must be long enough to allow signals to be valid for the required setup time before the next clock edge (COD Figure A.11.1).

The time for a flip-flop input to propagate to the flip-flop outputs is t_{prop}; the signal then takes $t_{combinational}$ to travel through the combinational logic and must be valid t_{setup} before the next clock edge.



One additional complication that must be considered in edge-triggered designs is *clock skew*. Clock skew is the difference in absolute time between when two state elements see a clock edge. Clock skew arises because the clock signal will often use two different paths, with slightly different delays, to reach two different state elements. If the clock skew is large enough, it may be possible for a state element to change and cause the input to another flip-flop to change before the clock edge is seen by the second flip-flop.

Clock skew: The difference in absolute time between the times when two state elements see a clock edge.

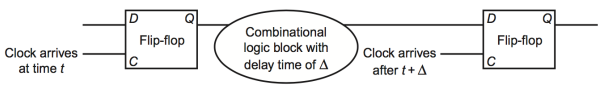
The figure below illustrates this problem, ignoring setup time and flip-flop propagation delay. To avoid incorrect operation, the clock period is increased to allow for the maximum clock skew. Thus, the clock period must be longer than

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}} + t_{\text{skew}}$$

With this constraint on the clock period, the two clocks can also arrive in the opposite order, with the second clock arriving t_{skew} earlier, and the circuit will work correctly. Designers reduce clock-skew problems by carefully routing the clock signal to minimize the difference in arrival times. In addition, smart designers also provide some margin by making the clock a little longer than the minimum; this allows for variation in components as well as in the power supply. Since clock skew can also affect the hold-time requirements, minimizing the size of the clock skew is important.

Figure 7.11.2: Illustration of how clock skew can cause a race, leading to incorrect operation (COD Figure A.11.2).

Because of the difference in when the two flip-flops see the clock, the signal that is stored into the first flip-flop can race forward and change the input to the second flip-flop before the clock arrives at the second flip-flop.



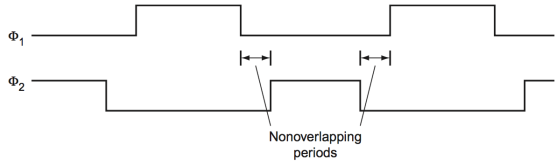
Edge-triggered designs have two drawbacks: they require extra logic and they may sometimes be slower. Just looking at the D flip-flop versus the level-sensitive latch that we used to construct the flip-flop shows that edge-triggered design requires more logic. An alternative is to use *level-sensitive clocking*. Because state changes in a level-sensitive methodology are not instantaneous, a level-sensitive scheme is slightly more complex and requires additional care to make it operate correctly.

level-sensitive clocking: A timing methodology in which state changes occur at either high or low clock levels but are not instantaneous as such changes are in edge-triggered designs.

Level-sensitive timing

In level-sensitive timing, the state changes occur at either high or low levels, but they are not instantaneous as they are in an edge-triggered methodology. Because of the noninstantaneous change in state, races can easily occur. To ensure that a level-sensitive design will also work correctly if the clock is slow enough, designers use *two-phase clocking*. Two-phase clocking is a scheme that makes use of two nonoverlapping clock signals. Since the two clocks, typically called φ_1 and called φ_2, are nonoverlapping, at most one of the clock signals is high at any given time, as the figure below shows. We can use these two clocks to build a system that contains level-sensitive latches but is free from any race conditions, just as the edge-triggered designs were.

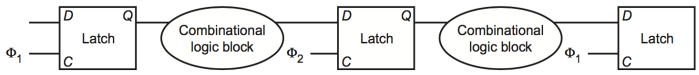
Figure 7.11.3: A two-phase clocking scheme showing the cycle of each clock and the nonoverlapping periods (COD Figure A.11.3).



One simple way to design such a system is to alternate the use of latches that are open on called φ_1 with latches that are open on φ_2. Because both clocks are not asserted at the same time, a race cannot occur. If the input to a combinational block is a φ_1 clock, then its output is latched by a φ_2 clock, which is open only during φ_2 when the input latch is closed and hence has a valid output. The figure below shows how a system with two-phase timing and alternating latches operates. As in an edge-triggered design, we must pay attention to clock skew, particularly between the two clock phases. By increasing the amount of nonoverlap between the two phases, we can reduce the potential margin of error. Thus, the system is guaranteed to operate correctly if each phase is long enough and if there is large enough nonoverlap between the phases.

Figure 7.11.4: A two-phase timing scheme with alternating latches showing how the system operates on both clock phases (COD Figure A.11.4).

The output of a latch is stable on the opposite phase from its C input. Thus, the first block of combinational inputs has a stable input during φ_2, and its output is latched by φ_2. The second (rightmost) combinational block operates in just the opposite fashion, with stable inputs during φ_1. Thus, the delays through the combinational blocks determine the minimum time that the respective clocks must be asserted. The size of the nonoverlapping period is determined by the maximum clock skew and the minimum delay of any logic block.



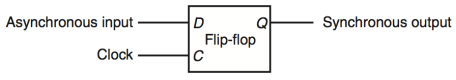
Asynchronous inputs and synchronizers

By using a single clock or a two-phase clock, we can eliminate race conditions if clock-skew problems are avoided. Unfortunately, it is impractical to make an entire system function with a single clock and still keep the clock skew small. While the CPU may use a single clock, I/O devices will probably have their own clock. An asynchronous device may communicate with the CPU through a series of handshaking steps. To translate the asynchronous input to a synchronous signal that can be used to change the state of a system, we need to use a *synchronizer*, whose inputs are the asynchronous signal and a clock and whose output is a signal synchronous with the input clock.

Our first attempt to build a synchronizer uses an edge-triggered D flip-flop, whose *D* input is the asynchronous signal, as the figure below shows. Because we communicate with a handshaking protocol, it does not matter whether we detect the asserted state of the asynchronous signal on one clock or the next, since the signal will be held asserted until it is acknowledged. Thus, you might think that this simple structure is enough to sample the signal accurately, which would be the case except for one small problem.

Figure 7.11.5: A synchronizer built from a D flip-flop is used to sample an asynchronous signal to produce an output that is synchronous with the clock (COD Figure A.11.5).

This "synchronizer" will *not* work properly!



The problem is a situation called *metastability*. Suppose the asynchronous signal is transitioning between high and low when the clock edge arrives. Clearly, it is not possible to know whether the signal will be latched as high or low. That problem we could live with. Unfortunately, the situation is worse: when the signal that is sampled is not stable for the required setup and hold times, the flip-flop may go into a metastable state. In such a state, the output will not have a legitimate high or low value, but will be in the indeterminate region between them. Furthermore, the flip-flop is not guaranteed to exit this state in any bounded amount of time. Some logic blocks that look at the output of the flip-flop may see its output as 0, while others may see it as 1. This situation is called a *synchronizer failure*.

Metastability: A situation that occurs if a signal is sampled when it is not stable for the required setup and hold times, possibly causing the sampled value to fall in the indeterminate region between a high and low value.

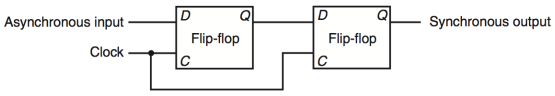
Synchronizer failure: A situation in which a flip-flop enters a metastable state and where some logic blocks reading the output of the flip-flop see a 0 while others see a 1.

In a purely synchronous system, synchronizer failure can be avoided by ensuring that the setup and hold times for a flip-flop or latch are always met, but this is impossible when the input is asynchronous. Instead, the only solution possible is to wait long enough before looking at the output of the flip-flop to ensure that its output is stable, and that it has exited the metastable state, if it ever entered it. How long is long enough? Well, the probability that the flip-flop will stay in the metastable state decreases exponentially, so after a very short time the probability that the flip-flop is in the metastable state is very low; however, the probability never reaches 0! So designers wait long enough such that the probability of a synchronizer failure is very low, and the time between such failures will be years or even thousands of years.

For most flip-flop designs, waiting for a period that is several times longer than the setup time makes the probability of synchronization failure very low. If the clock rate is longer than the potential metastability period (which is likely), then a safe synchronizer can be built with two D flip-flops, as the figure below shows. If you are interested in reading more about these problems, look into the references.

Figure 7.11.6: This synchronizer will work correctly if the period of metastability that we wish to guard against is less than the clock period (COD Figure A.11.6).

Although the output of the first flip-flop may be metastable, it will not be seen by any other logic element until the second clock, when the second D flip-flop samples the signal, which by that time should no longer be in a metastable state.



PARTICIPATION ACTIVITY 7.11.1: Check yourself: Clock skew.

1) Suppose we have a design with very large clock skew-longer than the register propagation time. Is it always possible for such a design to slow the clock down enough to guarantee that the logic operates properly?
   - ○ Yes
   - ○ No

Propagation time: The time required for an input to a flip-flop to propagate to the outputs of the flip-flop.

(*1) This section is in original form.

Provide feedback on this section