The Ten Commandments of Excellent Design

Peter Chambers
Engineering Fellow
VLSI Technology

This report will give you some pointers that will help you design synchronous circuits that work first time. Ten commandments that should *always* be followed!

Using Synchronous Circuits

Synchronous digital systems are pervasive in today's designs. Engineers create clocked circuits for every conceivable application, with frequencies from DC to GHz. Every synchronous system employs certain common characteristics, and is prone to a group of common faults. These faults can cause instability and unreliability, and may not be uncovered in the typical design process. The net result is a poor product that fails to meet the design criteria, and the engineer has to go through the suffering of design modification and revision. This is time-consuming and costly. However, by applying a few simple rules, you can avoid synchronous design faults in your designs and achieve consistent first-pass success. In this article you'll learn the sources of the most common problems and their solutions, and how to apply these ideas to your designs.

Digital Systems 101

We'll begin by describing a typical synchronous circuit. Many variations are possible, but a simple example will be adequate to illustrate the sources of error. Figure 1 shows the circuit and timing for one clocked element of the example.

One issue that deserves mention is this: Why use synchronous logic at all? Wouldn't asynchronous logic be faster? The answers to these questions could take a book, but here are some reasons to use synchronous designs:

- Synchronous designs eliminate the problems associated with speed variations through different paths of logic. By sampling signals at well-defined time intervals, fast paths and slow paths can be handled in a simple manner.
- Synchronous designs work well under variations of temperature, voltage and process. This stability is key for high-volume manufacturing.
- Many designs must be portable—that is, they must be easy to migrate to a new and improved technology (say, moving from .6 micron to .35 micron). The deterministic behavior of synchronous designs makes them much more straightforward to move to a new technology.
- Interfacing between two blocks of logic is simplified by defining standardized synchronous behavior. Asynchronous interfaces demand elaborate handshaking or token passing to ensure integrity of information; synchronous designs with known timing characteristics can guarantee correct reception of data.

Heck, I Know What a Flip-Flop Is!

Synchronous circuits are made with a mixture of combinatorial logic and clocked elements, such as flip flops or registers. The clocked elements share a common clock, and all transition from one state to another on the rising edge of

the clock. When the rising edge occurs, the registers propagate the logic levels at their D inputs to their Q outputs.
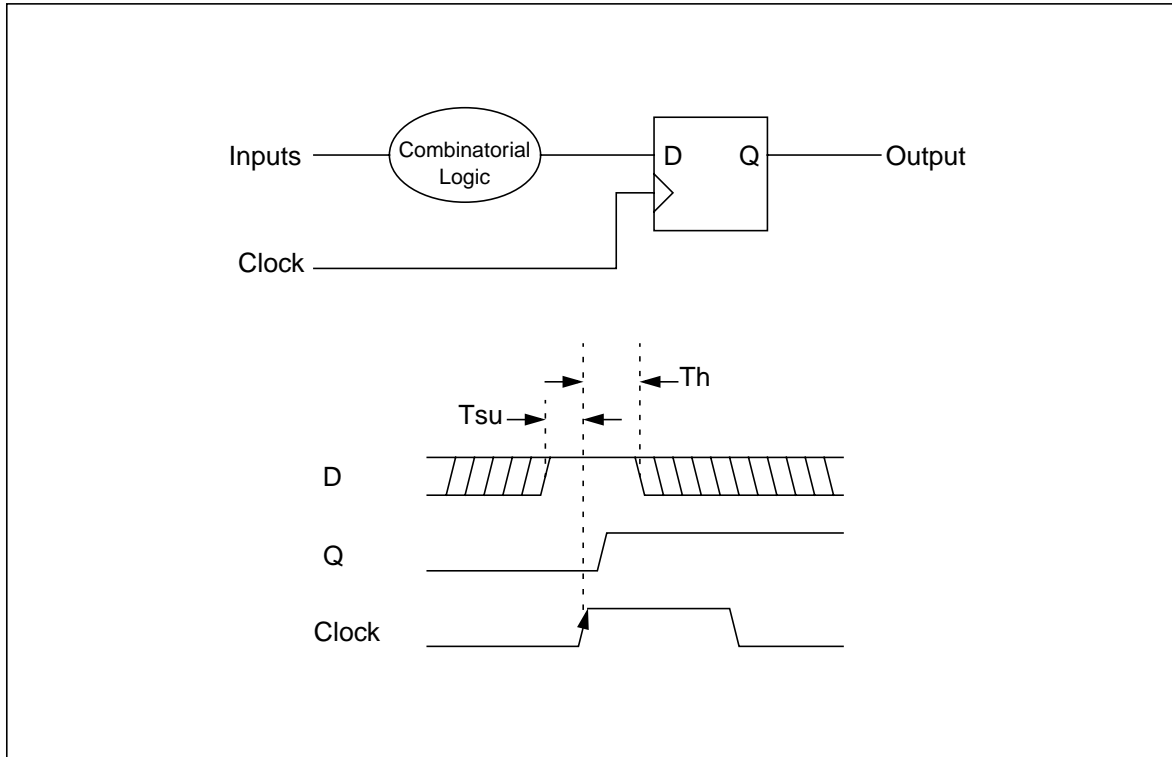

FIGURE 1. Simple Example of a Synchronous Circuit

In Figure 1, two important timing parameters are defined:

- Setup Time—Tsu
  Setup time is the time that the D input to a register must be valid before the clock transitions.
- Hold Time—Th
  Hold time is the period that the D input to a register must be maintained valid after the clock has transitioned.

If the setup or hold time parameters are violated terrible things happen. We'll discuss this later in the section on synchronization.

Clock Distribution (Yawn)

The distribution of clocks throughout a design has received considerable attention with the increase in logic speed. Common-or-garden personal computers have bus speeds of 66 MHz, and processor clocks run at 300 MHz or greater. In this article we're concerned more with the possible pitfalls in the synchronous logic itself, not with the production of decent clocks. However, for completeness, here are the important parameters necessary for a good clock distribution system design:

- Skew Minimization
  Clock skew is the variation in time of the clock's active transition being detected by different devices within a system. Skew must be kept to a minimum to ensure that setup and hold times are not violated at any one device. Methods for managing skew include equal-length traces, zero-delay PLL-based buffers, and additional logic for extending hold times.

- Clock Fidelity
  The clock's waveform must be as clean and deterministic as possible. Techniques used to guarantee consistent clock behavior include transmission line termination, ground-bounce minimization, and the use of identical clock drivers.

Good State Machine Design

One of the designer's most powerful constructs for synchronous design is the state machine. Combining combinatorial logic and a number of registers, the state machine is capable of making decisions based on its inputs and its current state. The behavior of the state machine is entirely synchronous, with all decisions taken at the time of the clock transition. There are two conventional forms of state machine: Mealy and Moore. The characteristics of these machines are shown in Figure 2.
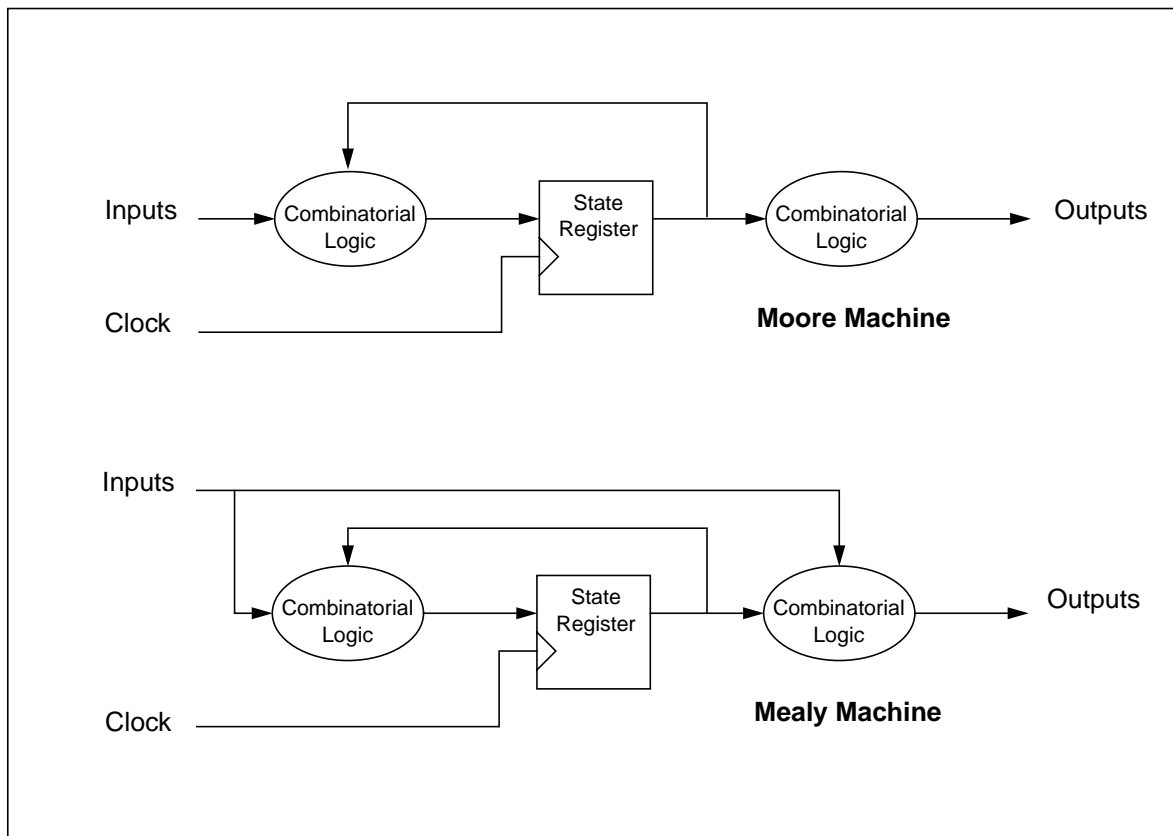

FIGURE 2. Characteristics of Mealy and Moore Machines

- Moore Machines
  Moore machines are the simpler of the two standard types. The output is a function only of the current state of the machine.

- Mealy Machines
  The outputs of Mealy machines are a function of the current state of the machine plus the inputs. This additional path provides more flexibility, but may complicate the understanding of the machine.

Books on high-level design languages (HDLs) expound at great length on the construction of state machines. The results are frequently disappointing. If you define your state machine in an HDL and run your design through a synthesizer, you may find spaghetti logic that no self-respecting designer would ever put together.

What's Wrong with Mealy/ Moore?

Figure 2 shows that the outputs of both the Mealy and Moore forms of state machine are combinatorial decodes of the current state and, in the Mealy form, the inputs. While this is fine in principle, there are pitfalls here waiting to trap the unwary.

The outputs of the state machine may include the following types of function:

- Latch enables (low- or high-going pulses to open or close latches)
- Tristate enables (signals to turn on and off drivers onto on-chip or off-chip buses)
- Register enables (enables to synchronously clocked registers)
- Other general control signals, such as counter enables, flags, and so on.

Most of these signals have one characteristic in common—glitches are absolutely unacceptable at any time. As the state registers and inputs of the Mealy or Moore state machines transition and settle, the combinatorial gates are quite capable of generating glitches as a consequence of the varying gate propagation delays. These transitory glitches may well contain enough energy to open latches, clock registers, and other highly undesirable effects.

Wouldn't Gray Code Fix the Problem?

We all learn at an early age that gray code counters are wonderful since only one bit changes at a time. When fed to an asynchronous decoder, theory suggests that the outputs should settle to their new state without noise. Your author is suspicious of this when the implementation is created by synthesized logic; unclocked feed-forward paths might well negate the advantage of gray code.

There is, however, a greater challenge to the use of gray code. The sequence of transitions taken by a state machine as it does its stuff is likely to be quite elaborate; many state machines are very complex with many branches between the possible states. Since gray code-driven decodes are only glitch free when a single bit changes at each clock edge, the designer must assure that all possible state transitions result in only a single bit change of the state variable. This is practical in only the simplest of state machines.

A Much Better State Machine

Figure 3 shows a much better design for a state machine. By adding an output register (with cleanly clocked D-type flip-flops) that is reloaded at each clock edge, the outputs of the state machine are guaranteed to be always glitch-free.

It is suggested that all state machines be implemented in this form, since the quality of the outputs is independent of the number of states or outputs.
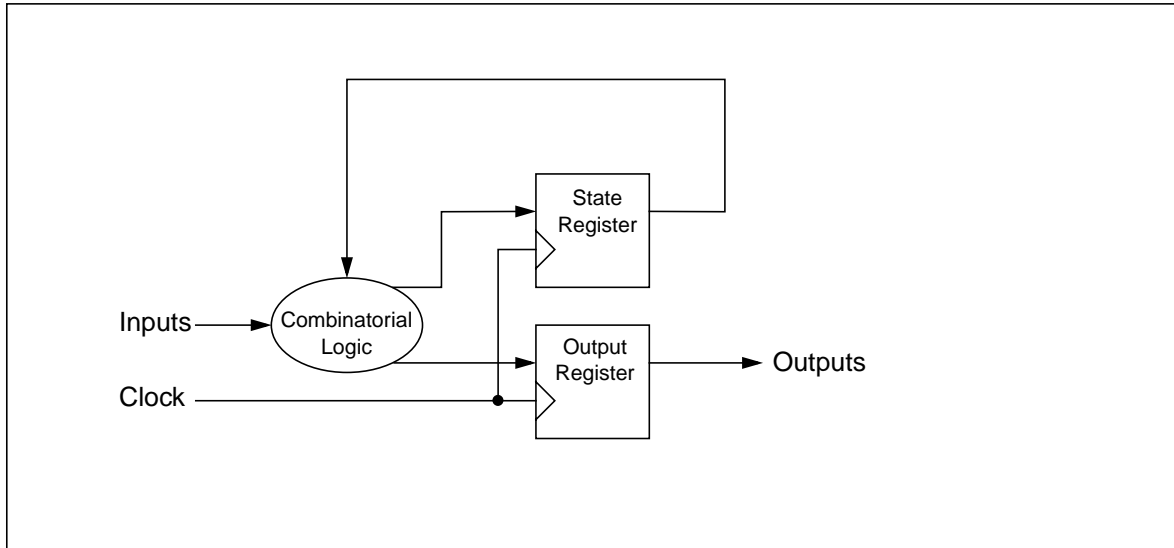
Feeding Inputs and Resets to Your State Machine

Reset signals are traditionally asynchronous and are routed directly to the clear inputs of state machine register elements. When the reset is asserted, all registers (state and output bits) are cleared immediately. All well and good, but what happens when the reset is deasserted? Consider a state machine that will transition from the reset state to some other state directly after the reset is deasserted. If the reset deasserts close to a clock edge, some of the state bits will assume their new states, while others might not. The state machine ends up in an undefined error state, and, yet again, you have egg on your face.

The solution? *Synchronize* that darned reset! That way, the reset will be removed well before the clock edge, and all register elements will correctly transition to their new states.

Synchronize *All* State Machine Inputs

In fact, every input to your state machine must be synchronous. At the very least, you must be absolutely certain that no input will violate the setup and hold times of the state machine's state and output registers.

Dead States—The Purgatory of State Machines

State machines with encoded state bits don't always use all possible states. For example, if you have a 20-state state machine, you would use a five-bit state register. This would leave 12 unused state values. Since states are usually counted incrementally from zero, our example would look like this:

States	What The States are Used For
0-19	Normal operation.
20-31	Not used: these are "dead" states.

If the state machine ever enters a state 20-31, errors are likely; worse, the machine may lock up totally, with the state machine forever in one of these illegal states. It may require a hard reset to recover from this condition.

Clearly, it's best to ensure your state machine never reaches a dead state. However, a robust design will at a minimum ensure that if the state machine does enter a dead state, it will exit the dead state immediately and then perhaps enter a quiescent state.

Crossing Clock Domains

Moving information from one clock domain to another is rather like descending into Dante's inferno. All sorts of evils lie in wait to beset the naive. Setup and hold violations, metastability conditions, unreliable data, and other perils are manifest when moving from one clock domain to another. Indeed, the whole issue of synchronization might merit its own article. Here, a few tips will be presented which might help in resolving the block-to-block synchronization issues.
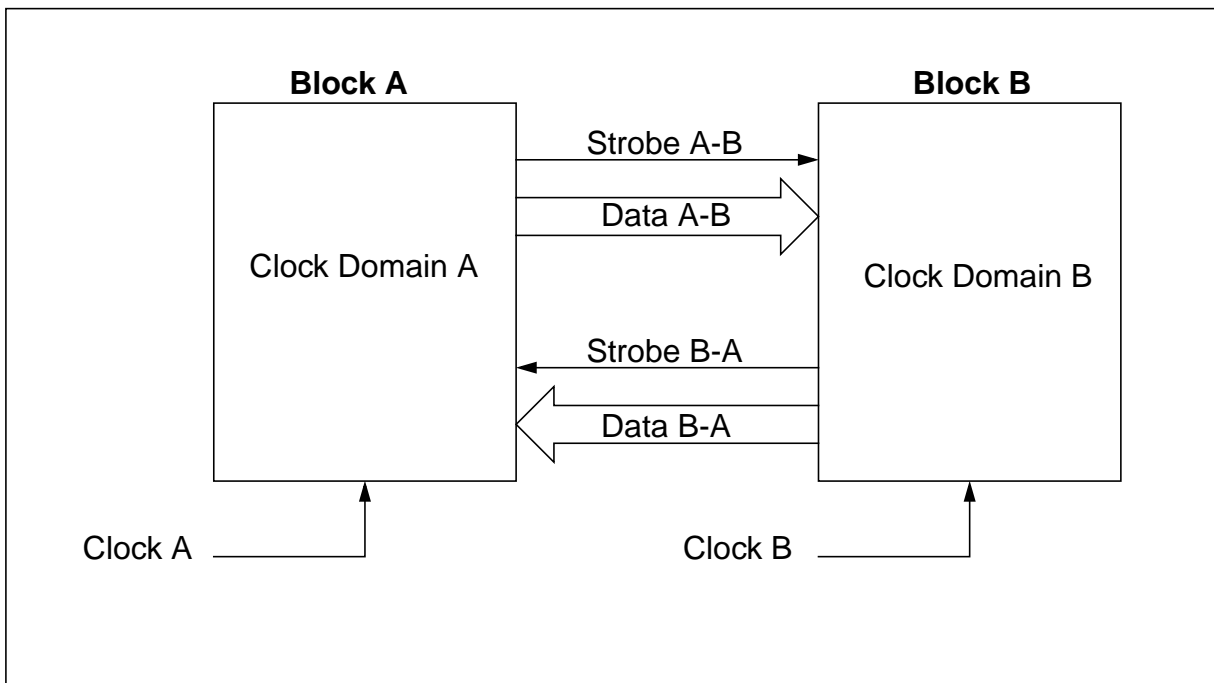
First, let's define the problem; please see Figure 4.

FIGURE 4. **Crossing Clock Domains**

We have two blocks of logic, A and B. Block A operates with Clock A, while Block B operates with Clock B. We make no assumptions at all about the frequencies of Clock A and Clock B; nor do we assume any integer or multiple relationship between the two. The two clocks are totally independent.

We need to send a strobe from Block A to Block B (Strobe A-B), and also some data, Data A-B. In response, Strobe B-A returns, together with Data B-A. The transmission of information between the blocks must be absolutely reliable. To accomplish this, we will look at several aspects of the cross-domain problem.

Synchronization 101

Crossing between clock domains is a similar issue to managing asynchronous inputs. Since no relationship between the multiple clock domains can be assumed, the inputs from Block A to Block B must be assumed to be asynchronous inputs. The traditional way of synchronizing an asynchronous input signal is shown in Figure 5:
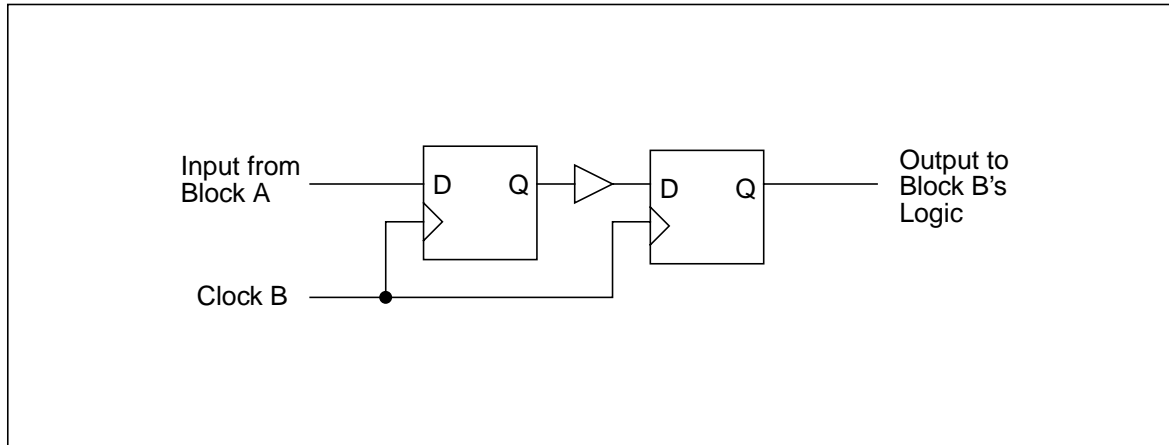

FIGURE 5. **Synchronizing an Asynchronous Input**

Two D-type flip-flops are used; two synchronization stages are usually sufficient. Only the rarest applications might demand three stages of synchronization. If your silicon library supports metastable-hardened flip-flops, then the first stage should use such a device. Typically, metastable-hardened flip-flops guarantee that their Q outputs will settle after a given maximum time, no matter how close the data transition is to the flip-flop's clock edge.

This method of information interchange has one drawback. If the strobe has the form of a pulse, it may not be seen by the destination block if the pulse width is less than the destination block's clock (sampling) frequency. This is not a problem if the two blocks exchange levels instead of pulses; however, this is slow, as typically four level exchanges must occur for a two-way handshake. The toggle method described later is an excellent solution to this problem.

Single-Point Information

Imagine that Block A needs to send two bits of information to Block B. We could simply duplicate the circuit in Figure 5, with one synchronization circuit for each bit. There is a serious problem which should be clear: occasionally, the circumstance will arise when one bit gets through the two-stage synchronization circuit, while the other does not. The result is ambiguous information and errors. The solution is shown back in Figure 4—use a *single* strobe from Block A to Block B, and send the rest of the information separately. The single-point strobe from A to B informs the destination block that the Data A-B is valid; the originating block ensures that there is adequate setup time.

Toggleo, Toggleas, Toggleat   A nifty way of doing a two-way handshake without worrying about levels and
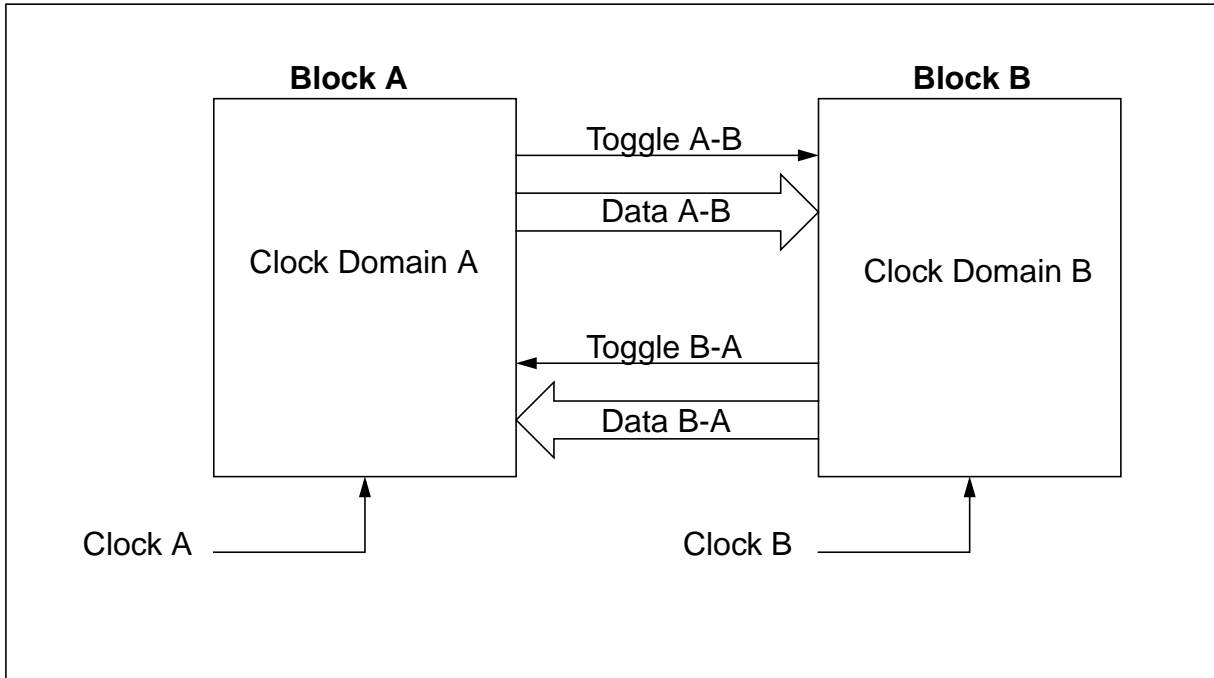pulse widths is to use a toggle exchange protocol. This is illustrated in Figure 6.

FIGURE 6. **Using Toggle Signals to Cross Clock Domains**

In this case, the signal from Block A to Block B that indicates the data (Data A-
B) is valid is a *transition* of the signal Toggle A-B. This transition may be low-to-

high or high-to-low. Both transitions have the same meaning: the Data A-B bus is valid. This is illustrated in Figure 7
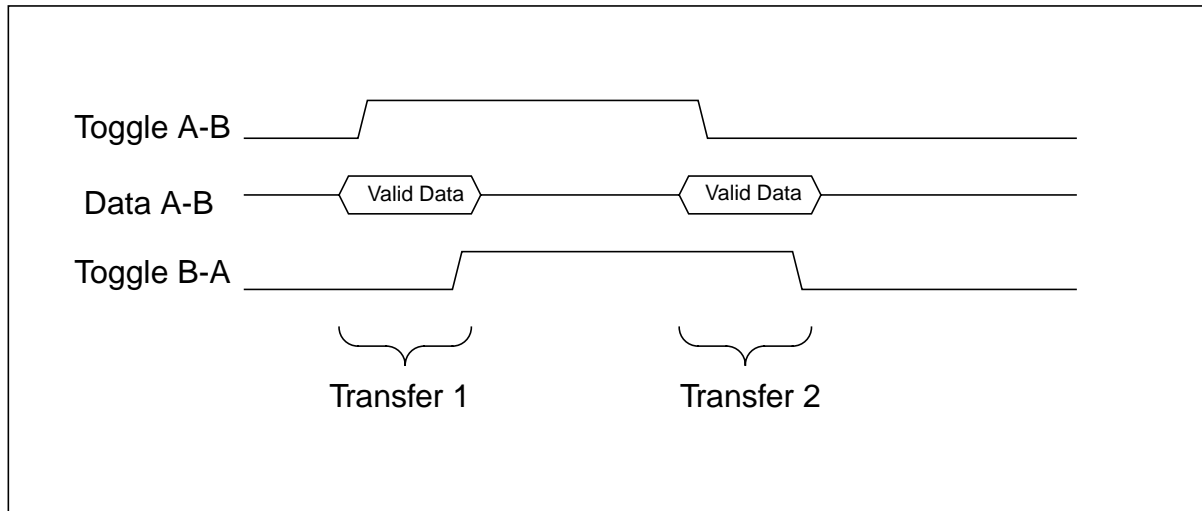
FIGURE 7. **Toggle Signal Timing: One Edge Does It All**

It may be seen that each transfer is complete with only two events: a toggle of each of the two Toggle strobes. While each toggle must, of course, be synchronized carefully at the receiving end, this method guarantees successful transmission and reception of wide data busses across clock domains of arbitrary frequency. From gigahertz to kilohertz, the toggle method is predictable and reliable.

Latches Look Lovely!

When creating a set of clocked elements, there is often a compelling reason to use latch-based designs. A single-bit register implemented with a latch may use just 60% of the gates that a conventional D-type flip-flop requires. If your design uses great numbers of configuration registers, FIFOs, or has elaborate data paths, the savings when using latches might be considerable. And since the latch control might be the same signal as the clock enable to a D-type flip-flop with a clock enable, why not use latches? Look at Figure 8, which shows how a latch works.
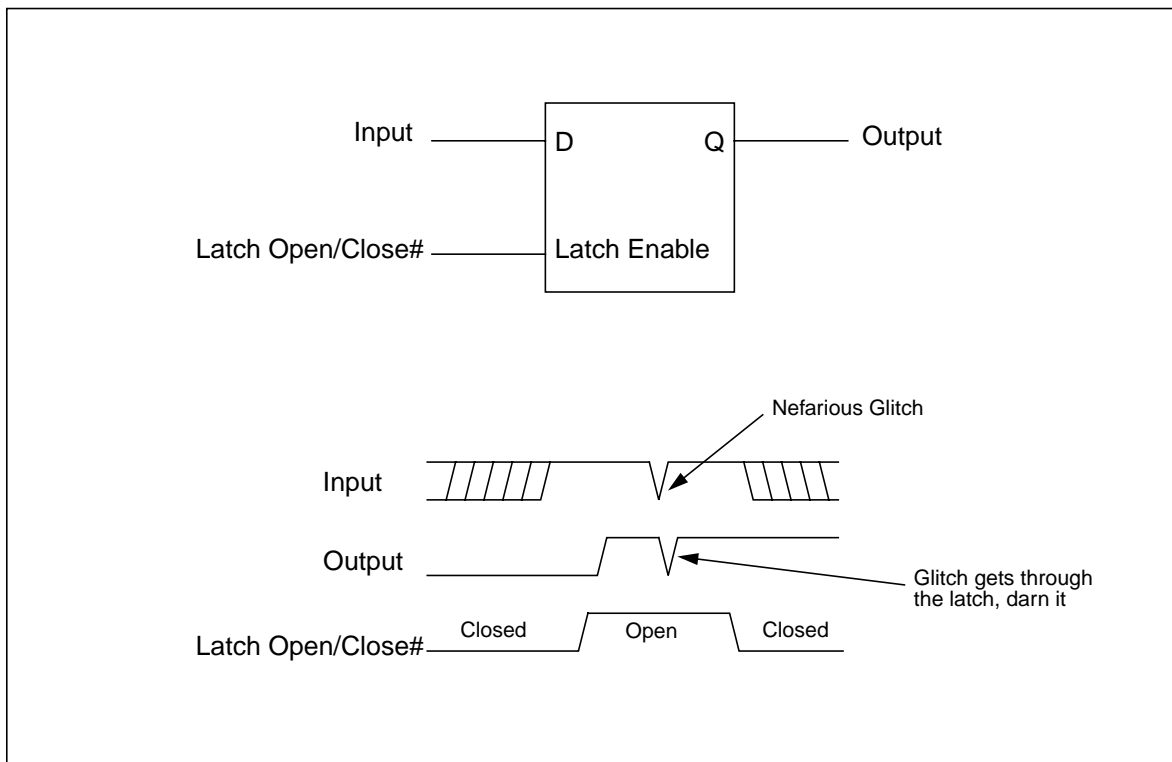
FIGURE 8. **How a Latch Works.**

The latch's Q output is stable while the latch is closed. When the latch is open, the input is continuously copied to the output. Two potential pitfalls exist with latches:

1. Noisy Inputs
   Any glitches on the latch's D input are propagated directly through to the output. This is, of course, manageable by ensuring that there aren't any glitches on the input. However, in a synchronous system, busses tend to switch states at clock edges, and the latch enable typically straddles a clock

edge, requiring that the D input be perfectly clean right through the same clock edge. This is the worst time for switching noise, particularly on wide busses. What's more, the latch needs the D input to be stable for two clock periods (so it's clean through the clock edge). If you change the D input with the same edge that closes the latch, you have a race which you're bound to lose (Murphy and his law, you know).

2. Noisy Latch Enable
    Perhaps worse than noise on latch inputs is noise on the enable line. If a latch enable glitches as a result of an asynchronous decode, your design is toast. The first part of this article discussed how to eliminate glitches on decoded signals; but if you get it wrong, a register-based design is still likely to be robust, since glitches on clock enables don't matter except when the clock transitions. Glitches on latch enables always mean instant death whenever they occur.

Registers Rule!

Register-based designs suffer from none of the disadvantages listed above. Race conditions are rare to non-existent, glitches on the control or D signals are unlikely to cause harm, and signals can be reliably latched in one clock period. A register-based design may be larger than its latch-based equivalent, but it will be more robust and will contribute toward first-silicon success.

Bottom line: If you absolutely have to use latches, beware!

The Fast Path to Disaster
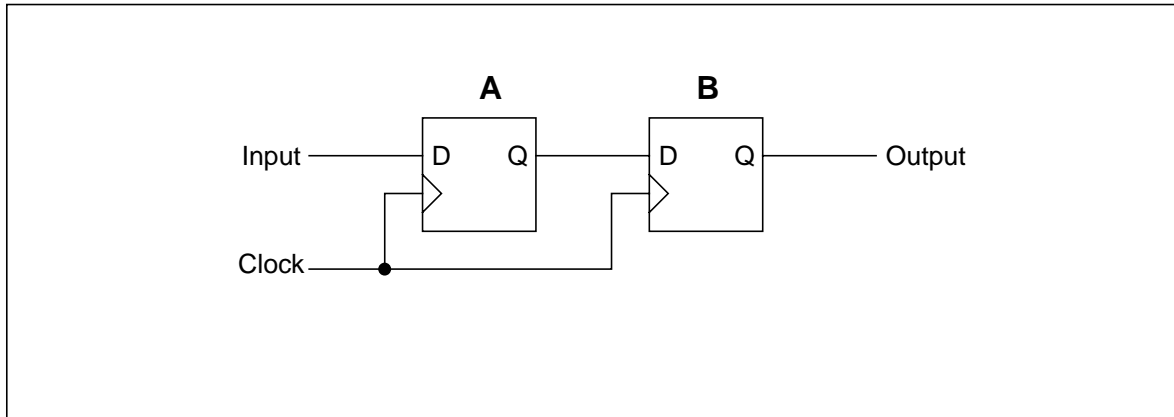
What's wrong with the circuit in Figure 9?

FIGURE 9. **Fast Paths and Race Conditions.**

This is a classic example of a race condition; the transition as the output of the first flip-flop changes might well violate the hold time on the D input of the second flip-flop. This situation can be worsened if there is skew between the clocks to each of the two flip-flops; if flip-flop B's clock lags A's, then B's output might actually replicate the output of A, rather than add the extra clock delay that is required. Figure 10 shows how to fix the problem:
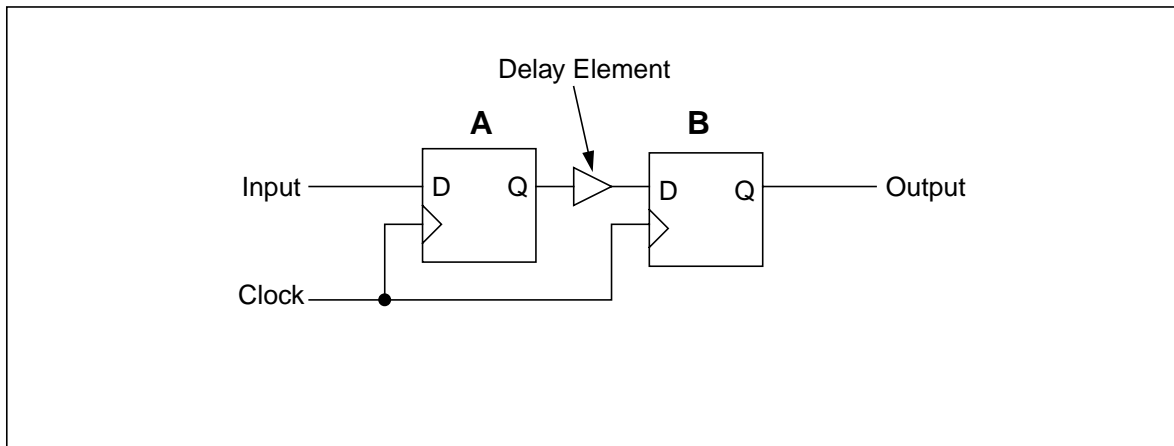

FIGURE 10. **After an Application of Fast Paths-B-Gone.**

The delay element ensures that there is sufficient time for flip-flop B to complete its transition before the result of A's transition reaches B.

Some synthesizer tools have a "fix hold" option which claims to take care of this situation. But if your design fails, who gets the blame: the designer or some well-hidden option in a synthesizer? Check carefully for fast paths.

Have Sympathy for the Test Engineer

If all goes well, your chip will enter mass production and the world will rejoice (or at least the shareholders). To do this, your design must be testable. Testability is a much-neglected aspect of many designs; here are a few tips to help test engineers sleep better at night.

- Break long counters into bite-size chunks
  Counters require lots of test vectors to ensure that all bits toggle correctly, and that carry bits are generated as they should be. To help keep the number of test vectors to a reasonable number, provide the ability to partition a counter into multiple smaller (for example, four bits each) counters. Then provide visibility of the most significant bit of each stage. That way the test sequence can verify that every counter stage works by observing the most significant bit's low-to-high and high-to-low transitions, and can reasonably conclude that the counter will work as a unit.

- Asynchronous feedback paths are a federal offense
  Even without considering the effect it has on a test engineer's disposition, logic that uses asynchronous feedback is generally bad for a number of reasons. It is hard to simulate, it may well be dependent on voltage, temperature, and process, it may be very susceptible to transients. Just as bad, it may be impossible to test on a fixed-frequency tester. If there are unclocked feedback paths in your design, make sure that they can be broken and analyzed from the tester. Better still, get rid of them altogether.

Simulators Seduce the Unwary...

It is easy and tempting to say "I'll just design it quickly, then find the bugs in simulation." This is a bad idea and is doomed from the start. Simulators are notorious for hiding the quirky details of your design. Examples include:

- Clock Synchronization
  Synchronizing flip-flops constantly battle metastability and glitching inputs. Their behavior is not even closely approximated by your average simulator; all you see is a clean transition at the clock edge. Crossing clock domains must always be correct by design from the earliest stages.

- Asynchronous Logic
  In a similar way, asynchronous logic is often simulated poorly. Certainly, fast paths and race conditions may be hidden. Some environments will determine (and optionally correct) hold-time violations, but this is not a universal panacea for correct asynchronous logic.

**Correct by Design
and
Correct by Inspection**

When designing logic that is outside the protected realm of clock-to-clock register-to-register implementations, the only solution for robust design is to do it right from the start. Your logic must be:

- Correct by Design
  Each gate, each line of VHDL or Verilog, must be understood completely. Don't hope that some set of simulations will find your bugs; you may neglect to test a part of your design, and if it was designed sloppily, it *will* fail.

- Correct by Inspection
  Disciplined layout will also make your design more robust, comprehensible, and maintainable. It should not be necessary to sort through a mass of ugly code or spaghetti gates to understand the operation of your function. Organized gates, commented code, and thorough accompanying documentation will provide a basis for a reliable design.

Peter's Provocative Pontifications— The Ten Commandments for Successful Design

1. **All state machine outputs shall *always* be registered**

2. **Thou shalt use registers, never latches**

3. **Thy state machine inputs, including resets, shall be synchronous**

4. **Beware fast paths lest they bite thine ankles**

5. **Minimize skew of thine clocks**

6. **Cross clock domains with the greatest of caution. Synchronize thy signals!**

7. **Have no dead states in thy state machines**

8. **Have no logic with unbroken asynchronous feedback lest the fleas of myriad Test Engineers infest thee**

9. **All decode logic must be crafted carefully—eschew asynchronicity**

10. **Trust not thy simulator—it may beguile thee when thy design is junk**

Latches, Schmatches

Since this material first appeared, the second commandment, *Thou shalt use registers, never latches*, has been somewhat controversial (to say the least). Dyed-in-the-wool latch users have been squealing that latches are wondrous things, and are the solution to good designs, compact chips, and peace on earth. Three clear advantages of latches are:

*   Considerably smaller than D-type flip-flops
*   Provide anticipation of the data (for example, the decode of a latched address can begin before the latch is closed)
*   Lower power, compared with continuously clocked flip-flops.

If you do insist on a latch-based design, watch out for the following:

*   A glitch-free enable—remember that glitches on the enable can corrupt the latch's data. If you are synthesizing the code to create the enable, consider seriously the direct instantiation of the gate that drives the enable to the latch. Don't trust optimized equations!
*   Data input hold time—ensure that the data is held for long enough as you close the latch. If your latch enable is derived from a clock, the latch will lag the clock, requiring the latch's D inputs to be held valid after the clock edge

Contact Information

Here's how to contact the author:

Peter Chambers

VLSI Technology, Inc.
8375 South River Parkway, M/S 250
Tempe, Arizona 85284

Phone: 602 752 6395

Email: peter.chambers@vlsi.com