**Convert Asynchronous Designs To Synchronous Equivalents**

Jul 6th, 1998

Practice Good Digital Design Techniques To Reap The Benefits Offered By Synchronous Logic.

[Contributing Author](https://www.electronicdesign.com/21809650)

The advent of programmable logic has given designers nearly infinite flexibility in implementing digital logic designs. However, with the flexibility afforded by these devices, some common design practices may get overlooked. Often, the speed in which today's tools can take a design from specification to implementation leads to numerous trial-and-error attempts to get the design to work. In haste, engineers might end up with a design that contains asynchronous circuits in places where synchronous circuits could be used. Because flexible parts are used to implement the logic, the designs easily fit to the parts with the asynchronous logic intact.

This article will discuss the advantages of synchronous circuits over asynchronous circuits, and will also present common rules for designing good synchronous circuits. Lastly, we'll look at some circuits that contain asynchronous behavior and provide better synchronous alternatives.

If there were but one rule to follow in digital design it would be to make the design completely synchronous. Using registers that are clocked by a single common clock leads to the best overall system designs for a variety of reasons.

First of all, synchronous designs are more reliable. They are deterministic in their behavior, due to the fact that all signals are sampled at a well-defined time interval. Synchronous designs rely on very few timing parameters to guarantee operation, namely, the maximum frequency of operation of a device (fmax), the register setup and hold times (tSU and tH), and the register clock-to-output time (tCO). Meeting these parameters ensures designs will work under temperature, voltage, and process variations.

Synchronous designs are also portable. In all PLDs and ASICs, the master clock, or clocks, are routed via a low-skew clock network. These networks ensure that a design done in one PLD architecture will be compatible with a different architecture, with good results. Synchronous designs take advantage of this trait.

In addition, synchronous designs can be tested more easily and run statically, with the clock input driven by a test signal. They can be made virtually immune to noise. Therefore, finding errors in a design will not be a cross between identifying logic errors and tracking down noise-induced errors.

Synchronous designs attain performance levels easily. The maximum operational frequency of a synchronous design can be determined from the data sheet for many PLDs. Determining maximum performance of circuits that include asynchronous clocking events is much more complicated.

Finally, synchronous designs are easier to code in a hardware description language (HDL), and are also easier to read. Designs built around a common clock yield compact, efficient code. On the other hand, designs with numerous clocks and asynchronous behavior are more difficult to understand. Their code descriptions can also get cumbersome.

Synchronous Rules To Live By  
All inputs to a synchronous circuit need to be synchronous. If an asynchronous input to a synchronous circuit violates the tSU or tH of the registers, some of the registers may resolve the input as a logic 1, while others resolve it to logic 0. The classic way to synchronize asynchronous signals is to drive the signal through two cascaded D flip-flops.

Most PLD architectures guarantee a very high mean time between failure (MTBF) with this type of circuit, up to the fmax specification on the device data sheet. The MTBF, in this sense, is a statistical value that measures how often, on average, the second register in the synchronizer will receive an input that is not yet resolved by the first register. For example, the Cypress Flash370i and Ultra37000 families of CPLDs guarantee a 10-year MTBF for this type of circuit. This circuit is designed into the input macrocells of the device. Thus, the output of the second register will provide a signal that is synchronous to the rest of the logic.

When a design relies on more than a single clock, and information needs to be transferred from one clock domain into the other, the interaction must be treated as an asynchronous event—unless there's a known phase and frequency relationship between the clock domains. If many signals need to be transferred, a single synchronized handshake signal from the source-clock domain to the receiving domain should be used. When the handshake signal is received, the remaining signals can be captured from the source side. Those remaining signals should be sent across without synchronization. This guarantees that they have enough settling time to meet the tSU for the clock in the other domain.

Every asynchronous external signal input to a finite state machine (FSM) must be synchronized (with the two flip-flop synchronizer) to the FSM's clock to ensure appropriate behavior. Imagine the chaos that would result if a signal were to be left asynchronous and, while the input was in transition, some of the FSM's state registers detected a logic low while others detected a logic high.

It's just as important to keep outputs from an FSM synchronized. FSM outputs can be used for such functions as counter enables, register enables, and output enables. In any of these cases, the signal integrity is best when kept synchronous with the FSM's clock. This prevents unwanted propagation delays and possible glitches when the FSM transitions between states.

In many cases, engineers insert buffers and inverters in their designs to create an artificial delay. Too often, these delays are used to fix bad design techniques, such as using a register output to drive the clock input of another register. The delay might be put in the clock path to ensure that data arrives at the register before the clock does. The problem with adding delays is that the delay time is always unpredictable. What works today might not work if the design is ported to another device. The temperature differs, the process used for the device changes, the version of the logic synthesizer changes, and so on.

Unfortunately, we often violate the aforementioned rules when time constraints and old habits overcome us. But, by combining knowledge of the these rules with the following tools for eliminating asynchronous circuits from designs, designers can achieve a high-performance, successful result that will be reliable, portable, and easy to test.

Lower frequency operation is often required in today's designs. Many designers simply take an output of a counter and use it as the clock to another synchronous circuit *(*[*Fig. 1*](https://www.electronicdesign.com/files/29/7596/figure_01.gif)*)*.

Using the counter output forces internal clocking within a PLD—an external global clock is not the clock source for some registers. Because the counter output is registered, it can be used reliably as a clock to another circuit without the possibility of glitches. However, there are three main problems associated with this type of circuit:

1. The timing of the circuit is more difficult to analyze and the circuit will not run at the device's fmax. This is because there's a clock-to-output delay (tCO) from CLK to /16, and another tCO from /16 to OUTx. If the OUTx signals are distributed to logic clocked by CLK, both tCO delays must be accounted for in the calculation of the circuit's fmax.
2. The timing problem is exacerbated if a ripple counter is used. For each output of the counter, an additional tCO needs to be added to the timing. In addition, the timing of the internal clocking (often called asynchronous clocks, or product-term clocks) and external global clocking will most likely be different, and require careful analysis.
3. The design may not fit within the target device architecture if internal clocking isn't supported. However, all devices have at least one global clock input that can be driven from a device pin.

This type of design can be converted to an equivalent circuit that is only dependent on a single clock. Notice that the CLK clock edge, where the counter changes from a value of 7 (0111) to 8 (1000), is ultimately when the OUTx signals are changed as well. Converting from using the /16 counter output as a clock to using the CLK signal as the clock requires that an enable signal be created. This lets the synchronous circuit operate only when the counter is transitioning from 7 to 8 *(*[*Fig. 2*](https://www.electronicdesign.com/files/29/7596/figure_02.gif)*)*.

The timing of this circuit is only tied to the transition of CLK. It will easily fit into any device because it uses only one clock, with no internal clocking. In a CPLD with a simple timing model (such as the Cypress Flash370i or Ultra37000 families), the circuit will run at the maximum frequency allowed by the device. In an FPGA or ASIC, the timing is easy to analyze. The enable signal is the only critical path, external to the synchronous circuit, for determining the maximum frequency.

The only issue that designers need to be concerned with is the additional logic required by the enable signal. The implementation of the enable is done with the logic of a 2:1 multiplexer on the D input to a register. The enable is the select line to the multiplexer, which chooses between the two inputs. One input to the multiplexer is the register's output, while the other is the logic to be implemented when the register is enabled. Thus, the register will either retain its prior value on a clock edge, or possibly change to a new value, depending on the logic input.

In a CPLD, the circuitry shown in Figure 2 will require three more inputs to a logic block for the additional counter outputs, an additional input to the logic block for each OUTx, and five product terms for each OUTx (this assumes that a single product term was originally required for a given OUTx's input equation). In a CPLD architecture, where the register can be configured to be a toggle flip-flop rather than a D flip-flop, the product term usage drops to two. However, FPGAs and ASICs have different logic architectures, and the enable logic may put more demands on the design. With appropriate logic synthesis, these demands are minimal.

This example illustrates a general technique of finding an equivalent, synchronous circuit to reproduce the operation of the asynchronous circuit. The easiest way to do this is to analyze the timing diagram of the asynchronous circuit, and determine when outputs transition. Then, create an enable signal that, when used in conjunction with a master clock, will cause the outputs to transition at the same time.

There's nothing inherently wrong with using the rising and falling edges of a clock in the same circuit. The problem lies in how the inverted clocks are created. Often, the use of both edges of a clock could speed up the operation of a circuit *(*[*Fig. 3*](https://www.electronicdesign.com/files/29/7596/figure_03.gif)*)*. Q1 and Q2 form a two flip-flip synchronizer, with the first register clocked off the rising edge of CLK and the second clocked off an inverted version of CLK. A few assumptions must be made about the characteristics of the CLK signal and registers. If the registers have a certain maximum frequency of operation (fmax), CLK must be run at fmax /2 if both edges are to be used (and possibly slower, as will be shown). We'll assume that the duty cycle of the CLK is 50%/50% to simplify the timing analysis.

If the logic shown is created with discrete TTL components, the propagation delay of the inverter (tINV) must be accounted for. To ensure that the tSU of Q3 isn't violated, the low time of the clock (tPWL) minus tINV must be greater than or equal to tSU. Thus tINV must be accounted for in determining the maximum rate at which CLK can be run.

If the logic is implemented in a CPLD or FPGA, the architecture of the device determines the timing requirements. In some architectures, CLK is driven directly to all of the device registers across a low-skew clock distribution network. However, to get an inverted version of CLK, the CLK signal might have to pass through a macrocell (in the case of a CPLD), or through a logic cell and routing (in the case of an FPGA).

The timing on tINV will vary, depending on how the clock inversion is created. In a CPLD, tINV is likely a single fixed value, regardless of how many registers need to be clocked with the inverted signal. But, in an FPGA, tINV will probably vary on a register-by-register basis due to placement of the registers and routing of the inverted clock.

Ultimately, the best-case scenario for timing analysis and device operation is for INVCLK to be created, with no skew in relationship to CLK (in other words, tINV = 0 ns). This can easily be done with an appropriate clock-buffering device. However, some FPGA and CPLD architectures, such as the Flash370i and Ultra37000 families, can create inverted versions of incoming clock signals. The device's registers can select either the inverted version of the clock or the original clock, and the clocks are generated with zero skew between them. Any delay from the clock pin to the registers is accounted for in the tSU and tH values for a register, and are equivalent for inverted and noninverted versions of the original clock.

Using both edges of a clock is a design practice than can speed up the operation of a circuit. Make sure you understand the implementation trade-offs when choosing to use this kind of circuit.

Asynchronous Bus Interface  
In each of the previous examples, there has been a periodic synchronous clock available. Some designs, however, have the clock inputs to registers and flip-flops clocked with internal signals, when no periodic clock is available as a reference. It's often possible to find an alternative circuit that would take advantage of global clocking of a device. By finding an alternative implementation of a circuit that uses a global clock, you can guarantee portability across many architectures and simplify timing analysis.

One common use of a programmable logic device is as a peripheral to a microcontroller or microprocessor with an asynchronous bus interface. Let's examine a register that is written to when a processor is accessing the address Axx0 *(*[*Fig. 4a*](https://www.electronicdesign.com/files/29/7596/figure_04.gif)*)*.

When the logic is reduced for fitting into a CPLD, the result is a fairly complex equation on the clock input to the register:

REGCLK = CSn + WRn + —A15 + A14

+ —A13 + A12 + A3 + A2 + A1 + A0

The timing diagram for this circuit indicates that the register actually changes value on the rising edge of the WRn signal. Also, the processor's data sheet shows that there is a guaranteed tSU and tH on the data bus around the WRn signal coming from the processor. With this in mind, it's possible to come up with an alternate circuit with WRn as the clock.

This is accomplished by extracting WRn out of the equation above, using it alone as a clock, and using the remaining term as an enable to the register *(*[*Fig. 4b*](https://www.electronicdesign.com/files/29/7596/figure_04.gif)*)*. Note that the polarity of the logic is inverted to account for an active high enable on the register. The operation is preserved, but the engineer now has a circuit that can be ported to any device with global clocking, rather than having to rely on a device with product-term clocks.

The previous example was a fairly simple example of extracting a clock out of an equation and using the remaining logic as an input to a register (in that case, an enable to a register). Consider an example where logic on six signals makes up the clock to a D flip-flop *(*[*Fig. 5*](https://www.electronicdesign.com/files/29/7596/figure_05.gif)*)*. A few simple steps will determine if the circuit in question can be implemented in an equivalent synchronous circuit that doesn't use product-term clocks.

First, look for signals that act like clocks. Typically, these are the signals with the shortest pulse periods (as WRn was in the previous example) or with periodic operation. They effect a change in an output around one edge. In Figure 5, WR1 and WR2 are examples of such signals. WR1 has a short pulse period, and its falling edge causes ALARM to go to 1. WR2 is a periodic signal that can cause a change on ALARM on its rising edge.

Next, look for signals that work in conjunction with the clocks to affect changes in the output. Obviously, A1 and A2 work in conjunction with WR1 to set ALARM, and A3 and A4 work with WR2 to set ALARM. These signals should then be extracted to independent registers *(*[*Fig. 6*](https://www.electronicdesign.com/files/29/7596/figure_06.gif)*)*.

For the register clocked by the falling edge of WR1, the logical OR of A1 and A2 can be used on the register's D input to produce its contribution to the final ALARM signal. Likewise, the logical OR of A3 and A4 can be used on the register's D input, clocked by the rising edge of WR2.

Independently, these registers produce a signal that can be used to determine the state of ALARM. In some cases, the logic might be created as a register enable combined with logic on the register's input. However, in this case, logic on the register's D input creates the proper signal. Both registers can use a common clear signal as an asynchronous reset. This will clear the intermediate states, and permit the registers to be ready for another clock edge.

In this example, the OR of the two registers produces the final output as shown in Figure 6. We have eliminated complexity from the clock input to a register, and created a circuit that can be implemented in any programmable logic device or ASIC—without requiring product-term clocking.

Following good design techniques is imperative when designing with the latest PLDs and ASICs. Creating a design is synchronous, and converting those that are not, will make it much easier to get a design to work properly the first time. By following these few simple conversion rules, an engineer can ensure the reliability, readability, portability, and testability of designs:

* Analyze the timing diagram and create an equivalent circuit with a single synchronous clock with clock enables.
* Understand how clocks are created, and the skews and delays that can be expected.
* Find a signal that looks like a clock. Extract it, and use the remaining logic as an enable to the register.
* Extract those same signals again. This time, use the remaining logic as input logic or enables for registers that are clocked by that particular clock. Combine the registers logically to achieve a final result.