1. Design Guidelines for HardCopy Series Devices

H51011-3.4

Introduction

HardCopy® series devices provide dramatic cost savings, performance improvement, and reduced power consumption over their programmable counterparts. In order to ensure the smoothest possible transfer from the FPGA device to the equivalent HardCopy series device, you must meet certain design rules while the FPGA implementation is still in progress. A design that meets standard, accepted coding styles for FPGAs, adheres easier to recommended guidelines. This chapter describes some common situations that you should avoid. It also provides alternatives on how to design in these situations.

Design Assistant Tool

The Design Assistant tool in the Quartus® II software allows you to check for any potential design problems early in the design process. The Design Assistant is a design-rule checking tool that checks the compiled design for adherence to Altera® recommended design guidelines. It provides a summary of the violated rules that exist in a design together with explicit details of each violation instance. You can customize the set of rules that the tool checks to allow some rule violations in your design. This is useful if it is known that the design violates a particular rule that is not critical. However, for HardCopy design, you must enable all of the Design Assistant rules. All Design Assistant rules are enabled and run by default in the Quartus II software when using the HardCopy Timing Optimization Wizard in the **HardCopy Utilities** (Project menu). The HardCopy Advisor in the Quartus II software also checks to see if the Design Assistant is enabled.

The Design Assistant classifies messages using the four severity levels described in Table 1–1.

Table 1–1. Design Assistant Message Severity Levels (Part 1 of 2)

Severity Level	Description
Critical	The rule violation described in the message critically affects the reliability of the design. Altera cannot migrate the design successfully to a HardCopy device without closely reviewing these violations.
High	The rule violation described in the message affects the reliability of the design. Altera must review the violation before the design is migrated to a HardCopy device.

Table 1–1. Design Assistant Message Severity Levels (Part 2 of 2)

Severity Level	Description
Medium	The rule violation described in the message may result in implementation complexity. The violation may impact the schedule or effort required to migrate the design to a HardCopy series device.
Information only	The message contains information regarding a design rule.

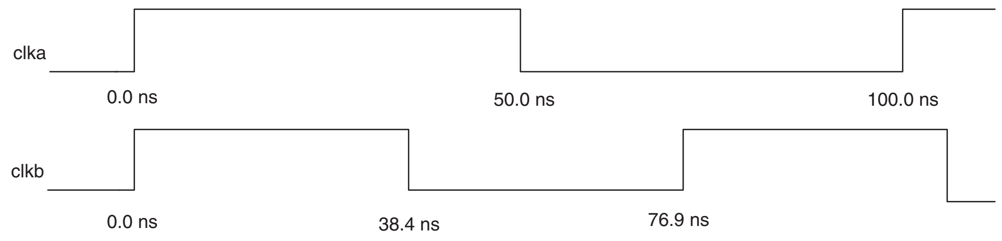
A design that adheres to Altera recommended design guidelines does not produce any critical, high, or medium level Design Assistant messages. If the Design Assistant generates these kinds of messages, Altera's HardCopy Design Center (which performs the migration) carefully reviews each message before considering implementing the FPGA design into a HardCopy design. After reviewing these messages with your design team, Altera may be able to implement the design in a HardCopy device. Informational messages are primarily for the benefit of the Altera HardCopy Design Center and are used to gather information about your design for the migration process from FPGA prototype to HardCopy production device.

Asynchronous Clock Domains

A design contains several clock sources, each driving a subsection of the design. A design subsection, driven by a single clock source is called a clock domain. The frequency and phase of each clock source can be different from the rest.

The timing diagram in Figure 1–1 shows two free-running clocks used to describe the nature of asynchronous clock domains. If the two clock signals do not have a synchronous, or fixed, relationship, they are asynchronous to each other. An example of asynchronous signals are two clock signals running at frequencies that have no obvious harmonic relationship.

Figure 1–1. Two Asynchronous Clock Signals *Notes (1), (2)*



Notes to Figure 1–1:
(1) `clka` = 10 MHz; `clkb` =13 MHz.
(2) Both clocks have 50% duty cycles.

In Figure 1–1, the `clka` signal is defined with a rising edge at 0.0 ns, a falling edge at 50 ns, and the next rising edge at 100 ns (1/10 MHz = 100 ns). Subsequent rising edges of `clka` are at 200 ns, 300 ns, 400 ns, and so on.

The `clkb` signal is defined with a rising edge at 0.0 ns, a falling edge at 38.45 ns, and the next rising edge at 76.9 ns. The subsequent rising edges of `clkb` are at 153.8 ns, 230.7 ns, 307.6 ns, 384.5 ns, and so on.

Not until the thousandth clock edge of `clkb` (1000 × 76.9 = 76,900 ns) or the 7,690th clock edge of `clka` (7,690 × 100 = 769,000 ns), does `clka` and `clkb` have coincident edges. It is very unlikely that these two clocks are intended to synchronize with each other every 76,900 ns, so these two clock domains are considered asynchronous to each other.

A more subtle case of asynchronous clock domains occurs when two clock domains have a very obvious frequency and phase relationship, especially when one is a multiple of the other. Consider a system with clocks running at 100 MHz and 50 MHz. The edges of one of these clocks are always a fixed distance away, in time, from the edges of the other clock. In this case, the clock domains may or may not be asynchronous, depending on what your original intention was regarding the interactions of these two clock domains.

Similarly, two clocks running at the same nominal frequency may be asynchronous to each other if there is no synchronization mechanism between them. For example, two crystal oscillators, each running at 100 MHz on a PC board, have some frequency variations due to temperature fluctuations, and this may be different for each oscillator. This results in the two independent clock signals drifting in and out of phase with each other.

Transferring Data between Two Asynchronous Clock Domains

If two asynchronous clock domains need to communicate with each other, you need to consider how to reliably perform this operation. The following three examples shows how to transfer data between two asynchronous clock domains.:

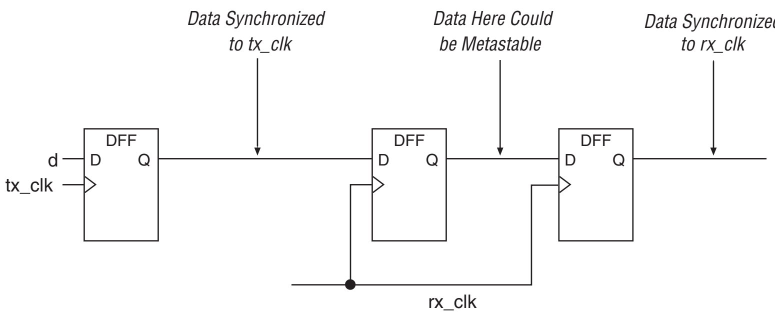
- Using a double synchronizer
- Using a first-in first-out (FIFO) buffer
- Using a handshake protocol

The choice of which to use depends on the particular application, the number of asynchronous signals crossing clock boundaries, and the resources available to perform the cross-domain transfers.

Using a Double Synchronizer for Single-Bit Data Transfer

Figure 1–2 shows a double synchronizer for single-bit data transfer consisting of a 2-bit shift register structure clocked by the receiving clock. The second stage of the shift register reduces the probability of metastability (unknown state) on the data output from the first register propagating through to the output of the second register. The data from the transmitting clock domain should come directly from a register. This technique is recommended only if single-data signals (for example, non-data buses) need to be transferred across clock domains. This is because it is possible that some bits of a data bus are captured in one clock cycle while other bits get captured in the next. More than two stages of the synchronizer circuit can be used at the expense of increased latency. The benefit of more stages is that the mean time between failures (MTBF) is increased with each additional stage.

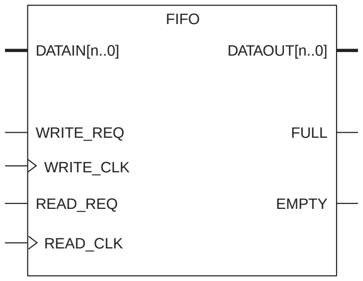
Figure 1–2. A Double Synchronizer Circuit


Data Synchronized to tx_clk
Data Here Could be Metastable
Data Synchronized to rx_clk
DFF
D
Q
d
tx_clk
DFF
D
Q
DFF
D
Q
rx_clk

Using a FIFO Buffer

The advantage of using a FIFO buffer, shown in Figure 1–3, is that Altera's MegaWizard® Plug-In Manager makes it very easy to design a FIFO buffer. A FIFO buffer is useful when you need to transfer a data bus signal across an asynchronous clock domain, and it is beneficial to temporary storage of this data. A FIFO buffer circuit should not generate any Design Assistant warnings unless an asynchronous clear is used in the circuit. An asynchronous clear in the FIFO buffer circuit results in a warning stating that a reset signal generated in one clock domain is not being synchronized before being used in another clock domain. This occurs because a dual-clock FIFO megafunction only has one `aclr` pin to reset the entire FIFO buffer circuit. You cannot remove this warning in the case of a dual-clock FIFO buffer circuit. As a safeguard, Altera recommends using a reset signal that is synchronous to the clock domain of the write side of the FIFO buffer circuit.

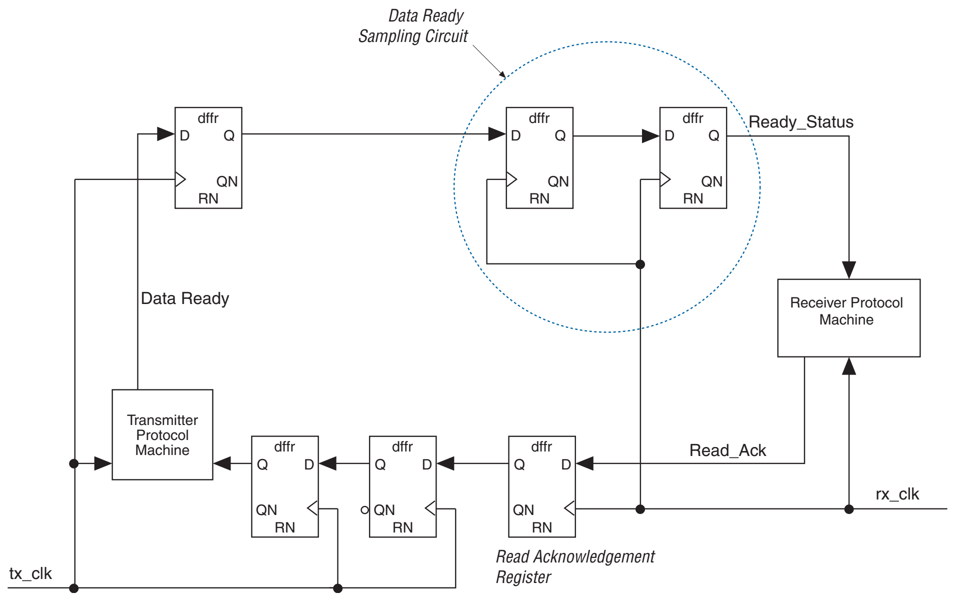
Figure 1–3. A FIFO Buffer



Using a Handshake Protocol

A handshake protocol circuit uses a small quantity of logic cells to implement and guarantee that all bits of a data bus crossing asynchronous clock domains are registered by the same clock edge in the receiving clock domain. This circuit, shown in Figure 1–4, is best used in cases where there is no memory available to be used as FIFO buffers, and the design has many data buses to transfer between clock domains.

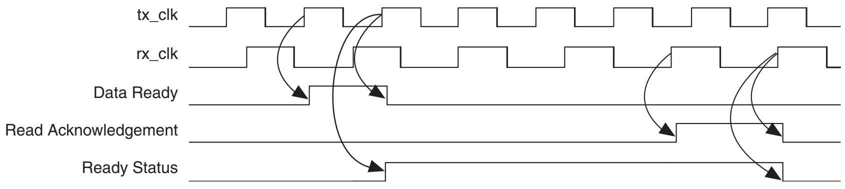
Figure 1–4. A Handshake Protocol Circuit


Data Ready Sampling Circuit
dffr
D
Q
QN
RN
Ready_Status
Data Ready
Receiver Protocol Machine
Transmitter Protocol Machine
Read_Ack
rx_clk
tx_clk
Read Acknowledgement Register

This circuit is initiated by a data ready signal going high in the transmitting clock domain `tx_clk`. This is clocked into the data ready sampling registers and causes the `Ready_Status` signal to go high. The `Data Ready` signal must be long enough in duration so that it is successfully sampled in the receiver domain. This is important if the `rx_clk` signal is slower than `tx_clk`.

At this point, the receiving clock domain `rx_clk` can read the data from the transmitting clock domain `tx_clk`. After this read operation has finished, the receiving clock domain (`rx_clk`) generates a synchronous `Read_Ack` signal, which gets registered by the read acknowledge register. This registered signal is sampled by the `Read_Ack` sampling circuit in the transmitter domain. The `Read_Ack` signal must be long enough in duration so that it is successfully sampled in the transmitter domain. This is important if the transmitter clock is slower than the receiver clock. After this event, the data transfer between the two asynchronous domains is complete, as shown by the timing diagram in Figure 1–5.

Figure 1–5. Data Transfer Between Two Asynchronous Clock Domains



Gated Clocks

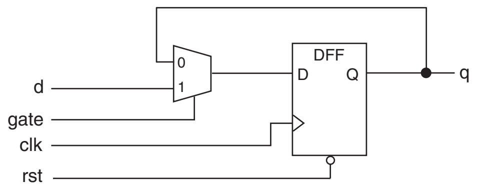
Clock gating is sometimes used to “turn off” parts of a circuit to reduce the total power consumption of a device. The gated clock signal prevents any of the logic driven by it from switching so the logic does not consume any power. This works best if the gating is done at the root of the clock tree. If the clock is gated at the leaf-cell level (for example, immediately before the input to the register), the device does not save much power because the whole clock network still toggles. The disadvantage in using this type of circuit is that it can lead to unexpected glitches on the resultant gated clock signal if certain rules are not adhered to. Rules are provided in the following subsections:

- Preferred Clock Gating Circuit
- Alternative Clock Gating Circuits
- Inverted Clocks
- Clocks Driving Non-Clock Pins
- Clock Signals Should Use Dedicated Clock Resources
- Mixing Clock Edges

Preferred Clock Gating Circuit

The preferred way to gate a clock signal is to use a purely synchronous circuit, as shown in Figure 1–6. In this implementation, the clock is not gated at all. Rather, the data signal into a register is gated. This circuit is sometimes represented as a register with a clock enable (CE) pin. This circuit is not sensitive to any glitches on the gate signal, so it gets generated directly from a register or any complex combinational function. The constraints on the gate or clock enable signal are exactly the same as those on the ‘d’ input of the gating multiplexer. Both of these signals must meet the setup and hold times of the register that they feed into.

Figure 1–6. Preferred Clock-Gating Circuit



This circuit only takes a few lines of VHDL or Verilog hardware description language (HDL) to describe.

The following is a VHDL code fragment for a synchronous clock gating circuit.

```
architecture rtl of vhdl_enable is
begin
    process (rst, clk)
    begin
        if (rst = '0') then
            q <= '0';
        elsif clk'event and clk = '1' then
            if (gate = '1') then
                q <= d;
            end if;
        end if;
    end process;
end rtl;
```

The following is a Verilog HDL code fragment for a synchronous clock gating circuit.

```
always @ (posedge clk or negedge rst)
    begin
        if (!rst)
            q <= 1'b0;
        else if (gate)
            q <= d;
        else
            q <= q;
    end
```

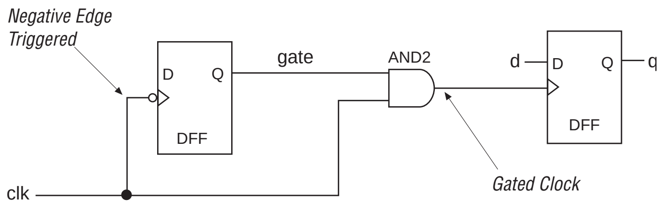
Alternative Clock Gating Circuits

If a clock gating circuit is absolutely necessary in the design, one of the following two circuits may also be used. The Design Assistant does not flag a violation for these circuits.

Clock Gating Circuit Using an AND Gate

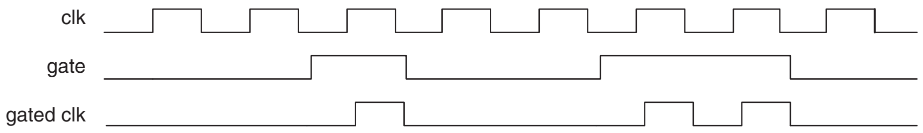
Designs can use a two-input AND gate for a gated clock signal that feeds into positive-edge-triggered registers. One input to the AND gate is the original clock signal. The other input to the AND gate is the gating signal, which should be driven directly from a register clocked by the negative edge of the same original clock signal. Figure 1–7 shows this type of circuit.

Figure 1–7. Clock Gating Circuit Using an AND Gate



Because the register that generates the gate signal is triggered off of the negative edge of the same clock, the effect of using both edges of the same clock in the design should be considered. The timing diagram in Figure 1–8 shows the operation of this circuit. The `gate` signal occurs after the negative edge of the clock and comes directly from a register. The logical AND of this `gate` signal, with the original un-inverted clock, generates a clean clock signal.

Figure 1–8. Timing Diagram for Clock Gating Circuit Using an AND Gate

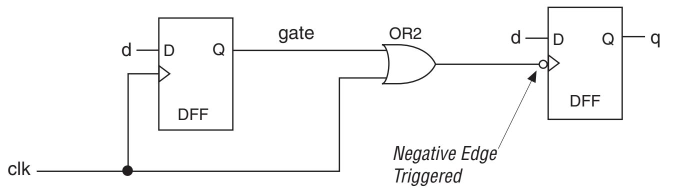


If the delay between the register that generates the `gate` signal and the `gate` input to the AND gate is greater than the low period of the clock, (one half of the clock period for a 50% duty cycle clock), the clock pulse width is narrowed.

Clock Gating Circuit Using an OR Gate

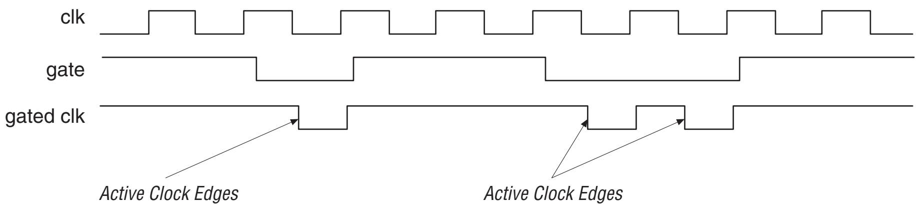
Use a two-input OR gate for a gated clock signal that feeds into a negative-edge-triggered register. One input to the OR gate is the original clock signal. The other input to the OR gate is the gating signal, which should be driven directly from a register clocked by the positive edge of the same original clock signal. Figure 1–9 shows this circuit.

Figure 1–9. Clock Gating Circuit Using an OR Gate



Because the register that generates the gate signal is triggered off the positive edge of the same clock, you need to consider the effect of using both edges of the same clock in your design. The timing diagram in Figure 1–10 shows the operation of this circuit. The `gate` signal occurs after the positive edge of the clock, and comes directly from a register. The logical OR of this `gate` signal with the original, un-inverted clock generates a clean clock signal. This clean, gated clock signal should only feed registers that use the negative edge of the same clock.

Figure 1–10. Timing Diagram for Clock Gating Circuit Using an OR Gate



If the delay between the register that generates the `gate` signal and the `gate` input to the AND gate is greater than the low period of the clock, (one half of the clock period for a 50% duty cycle clock), the clock pulse width is narrowed.

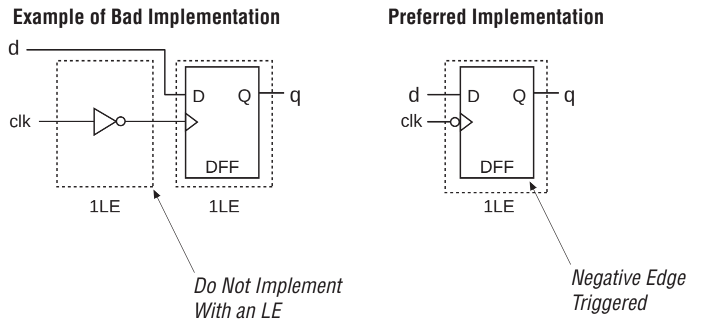
☞ Altera recommends using a synchronous clock gating circuit because it is the only way to guarantee the duty cycle of the clock and to align the clock to the data.

Inverted Clocks

A design may require both the positive edge and negative edge of a clock, as shown in Figure 1–11. In Altera FPGAs, each logic element (LE) has a programmable clock inversion feature. Use this feature to generate an inverted clock.

Do not instantiate a LE look-up-table (LUT) configured as an inverter to generate the inverted clock signal.

Figure 1–11. An LE LUT Configured as an Inverter



Using a LUT to perform the clock inversion may lead to a clock insertion delay and skew, which poses a significant challenge to timing closure of the design. It also consumes more device resources than are necessary. Refer to "Mixing Clock Edges" on page 1–14 for more information on this topic.

Do not generate schematics or register transfer level (RTL) code that instantiates LEs used to invert clocks. Instead, let the synthesis tool decide on the implementation of inverted clocks.

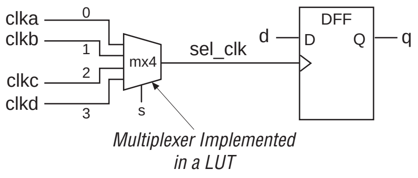
Clocks Driving Non-Clock Pins

As a general guideline, clock sources should only be used to drive the register clock pins. There are exceptions to this rule, but every effort should be taken to minimize these exceptions or remove them altogether.

One category of exception is for various gated clocks, which are described in "Preferred Clock Gating Circuit" on page 1–7.

You should avoid another exception, when possible, in which you use a clock multiplexer circuit to select one clock from a number of different clock sources, to drive non-clock pins. This type of circuit introduces complexity into the static timing analysis of HardCopy and FPGA implementations. For example, as shown in Figure 1–12, in order to investigate the timing of the `sel_clk` clock signal, it is necessary to make a clock assignment on the multiplexer output pin, which has a specific name. This name may change during the course of the design unless you preserve the node name in the Quartus II software settings. Refer to the Quartus II Help for more information on preserving node names.

Figure 1–12. A Circuit Showing a Multiplexer Implemented in a LUT



In the FPGA, a clock multiplexing circuit is built out of one or more LUTs, and the resulting multiplexer output clock may possibly no longer use one of the dedicated clock resources. Consequently, the skew and insertion delay of this multiplexed clock is potentially large, adversely impacting performance. The Quartus II Design Assistant traces clocks to their destination and, if it encounters a combinational gate, it issues a gated clock warning.

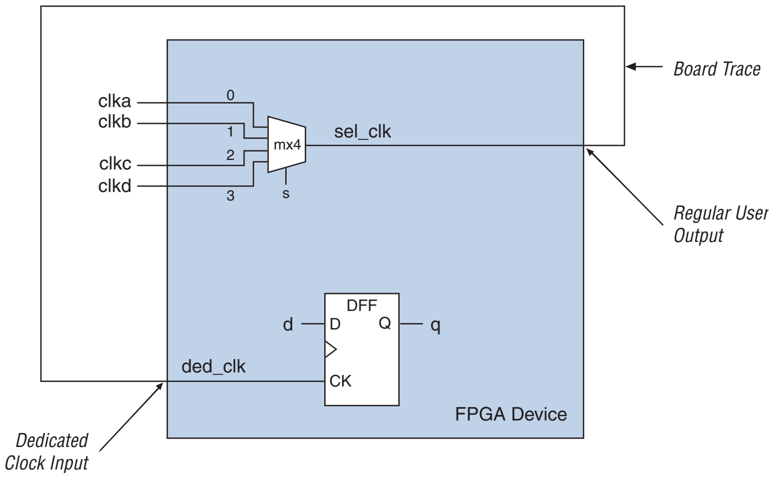
If the design requires this type of functionality, ensure that the multiplexer output drives one of the global routing resources in the FPGA. For example, this output should drive a fast line in an APEX™ 20KE device, or a global or regional clock in a Stratix® or Stratix II device.

Enhanced PLL Clock Switchover

Clock source multiplexing can be done using the enhanced PLL clock switchover feature in Stratix and Stratix II FPGAs, and in HardCopy Stratix and HardCopy II structured ASICs. The clock switchover feature allows multiple clock sources to be used as the reference clock of the enhanced PLL. The clock source switchover can be controlled by an input pin or internal logic. This generally eliminates the need for routing a multiplexed clock signal out to a board trace and bringing it back into the device, as shown in Figure 1–13.

Routing a multiplexed clock signal, as shown in Figure 1–13, is only intended for APEX 20K FPGA and HardCopy APEX devices. This alternative to a clock multiplexing circuit ensures that a global clock resource is used to distribute the clock signal over the entire device by routing the multiplexed clock signal to a primary output pin. Outside of the device, this output pin then drives one of the dedicated clock inputs of the same device, possibly through a phase-locked loop (PLL) to reduce the clock insertion delay. Although there is a large delay through the multiplexing circuit and external board trace, the resulting clock skew is very small because the design uses the dedicated clock resource for the selected clock signal. The advantage that this circuit has over the other implementations is that the timing analysis becomes very simple, with only a single-clock domain to analyze, whose source is a primary input pin to the APEX 20K FPGA or HardCopy APEX device.

Figure 1–13. Routing a Multiplexed Clock Signal to a Primary Output Pin



Clock Signals Should Use Dedicated Clock Resources

All clock signals in a design should be assigned to the global clock networks that exist in the target FPGA. Clock signals that are mapped to use non-dedicated clock networks can negatively affect the performance of the design. This is because the clock must be distributed using regular FPGA routing resources, which can be slower and have a larger skew than the dedicated clock networks. If your design has more clocks than are available in the target FPGA, you should consider reducing the number of clocks, so that only dedicated clock resources are used in the FPGA for clock distribution. If you need to exceed the number of dedicated clock resources, implement the clock with the lowest fan-out with regular (non-clock network) routing resources. Give priority to the fastest clock signals when deciding how to allocate dedicated clock resources.

In the Quartus II software, you can use the **Global Signal Logic** option to specify that a clock signal is a global signal. You can also use the auto **Global Clock Logic** option to allow the Fitter to automatically choose clock signals as global signals.

Altera recommends using the FPGA's built-in clock networks because they are pre-routed for low skew and for short insertion delay.

Mixing Clock Edges

You can use both edges of a single clock in a design. An example where both edges of a clock must be used in order to get the desired functionality is with a double data rate (DDR) memory interface. In Stratix II, Stratix, HardCopy II, and HardCopy Stratix devices, this interface logic is built into the I/O cell of the device, and rigorous simulation and characterization is performed on this interface to ensure its robustness. Consequently, this circuitry is an exception to the rule of using both edges of a clock. However, for general data transfers using generic logic resources, the design should only use a single edge of the clock. A circuit needs to use both edges of a single clock, then the duty cycle of the clock has to be accurately described to the Static Timing Analysis tool, otherwise inaccurate timing analysis could result. Figure 1–14 shows two clock waveforms. One has a 50% duty-cycle, the other has a 10% duty cycle.

Figure 1–14. Clock Waveforms with 50% and 10% Duty Cycles

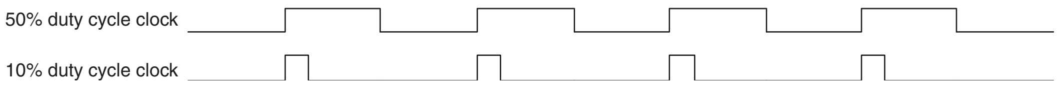


Figure 1–15 shows a circuit that uses only the positive edge of the clock. The distance between successive positive clock edges is always the same; for example, the clock period. For this circuit, the duty cycle of the clock has no effect on the performance of the circuit.

Figure 1–15. Circuit Using the Positive Edge of a Clock

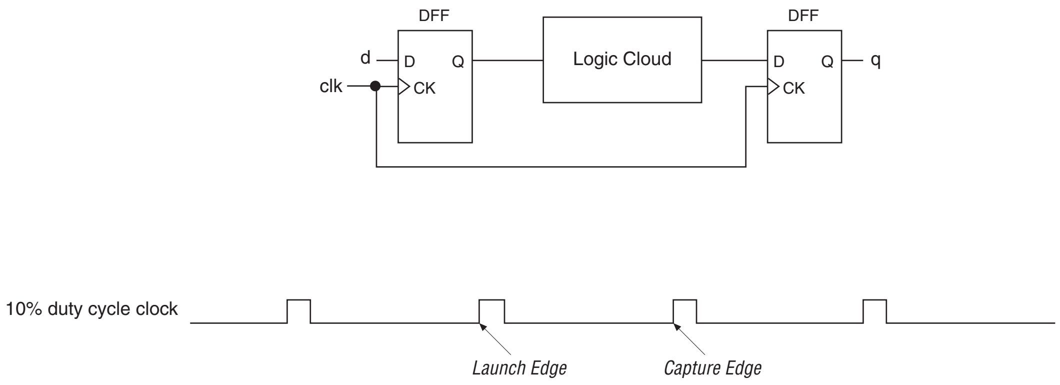
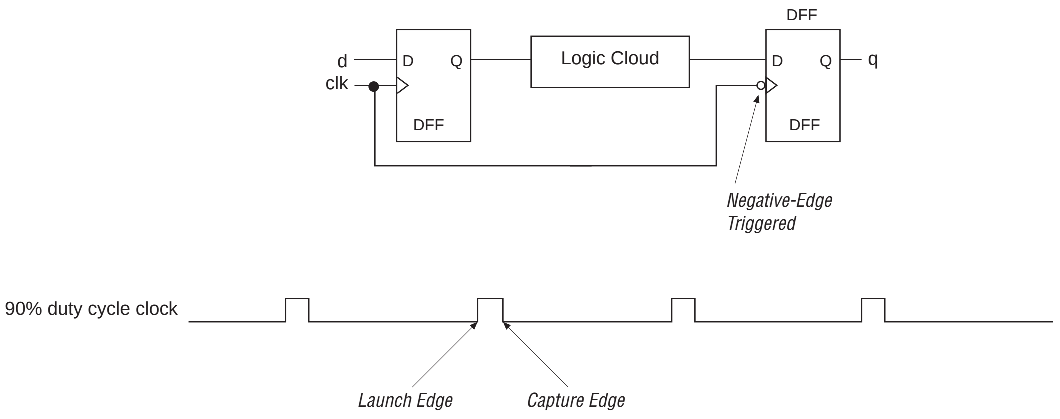


Figure 1–16 shows a circuit that used the positive clock edge to launch data and the negative clock edge to capture this data. Since this particular clock has a 10% duty cycle, the amount of time between the launch edge and capture edge is small. This small gap makes it difficult for the synthesis tool to optimize the cloud of logic so that no setup-time violations occur at the capture register.

Figure 1–16. Circuit Using the Positive and Negative Edges of a Clock



If you design a circuit that uses both clock edges, you could get the Design Assistant warning "Registers are Triggered by Different Edges of Same Clock." You do not get this warning under the following conditions:

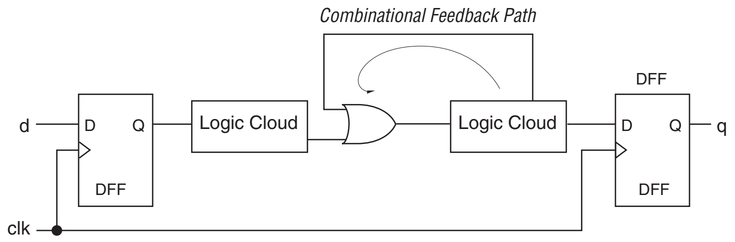
- If the opposite clock edge is used in a clock gating circuit
- A double data rate memory interface circuit is used

☞ Try to only use a single edge of a clock in a design.

Combinational Loops

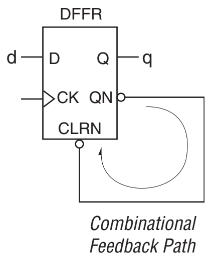
A combinational loop exists (Figure 1–17) if the output of a logic gate (or gates) feeds back to the input of the same gate without first encountering a register. A design should not contain any combinational loops.

Figure 1–17. A Circuit Using a Combinational Loop



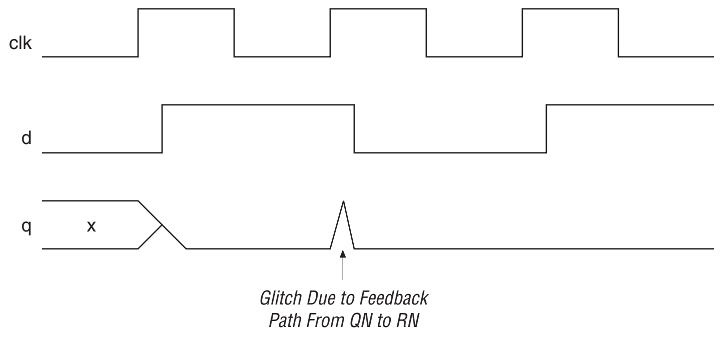
It is also possible to generate a combinational loop using a register (Figure 1–18) if the register output pin drives the reset pin of the same register.

Figure 1–18. Generation of a Combinational Loop Using a Register



The timing diagram for this circuit is shown in Figure 1–19. When a logic 1 value on the register D input is clocked in, the logic 1 value appears on the Q output pin after the rising clock edge. The same clock event causes the QN output pin to go low, which in turn, causes the register to be reset through RN. The Q register output consequently goes low. This circuit may not operate if there isn't sufficient delay in the QN-to-RN path, and is not recommended.

Figure 1–19. Timing Diagram for the Circuit Shown in Figure 1–18



Combinational feedback loops are either intentionally or unintentionally introduced into a design. Intentional feedback loops are typically introduced in the form of instantiated latches. An instantiated latch is an example of a combinational feedback loop in Altera FPGAs because its function has to be built out of a LUT, and there are no latch primitives in the FPGA logic fabric. Unintentional combinational feedback loops usually exist due to partially specified IF-THEN or CASE constructs in the register transfer level (RTL). The Design Assistant checks your design for these circuit structures. If any are discovered, you should investigate and implement a fix to your RTL to remove unintended latches, or re-design the circuit so that no latch instantiation is required. In Altera FPGAs, many registers are available, so there should never be any need to use a latch.

Combinational loops can cause significant stability and reliability problems in a design because the behavior of a combinational loop often depends on the relative propagation delays of the loop's logic. This combinational loop circuit structure behaves differently under different operation conditions. A combinational loop is asynchronous in nature, and EDA tools operate best with synchronous circuits.

A storage element such as a level-sensitive latch or an edge-triggered register has particular timing checks associated with it. For example, there is a setup-and-hold requirement for the data input of an edge-triggered register. Similarly, there is also a setup-and-hold timing requirement for the data to be stable in a transparent latch when the gate signal turns the latch from transparent to opaque. When latches are built out of combinational gates, these timing checks do not exist, so the static timing analysis tool is not able to perform the necessary checks on these latch circuits.

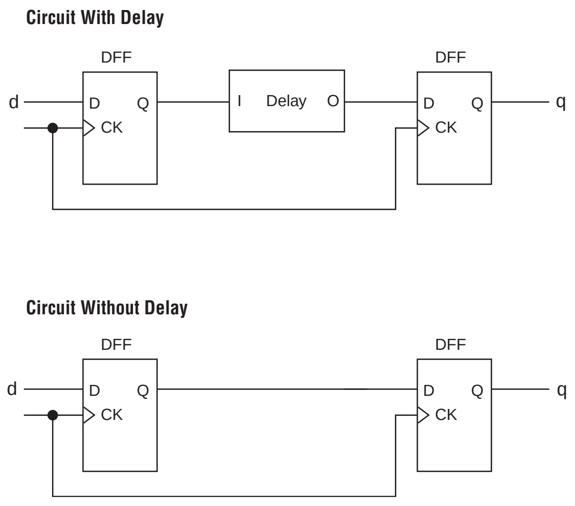
Check your design for intentional and unintentional combinational loops, and remove them.

Intentional Delays

Altera does not recommend instantiating a cell that does not benefit a design. This type of cell only delays the signal. For a synchronous circuit that uses a dedicated clock in the FPGA (Figure 1–20), this delay cell is not needed. In an ASIC, a delay cell is used to fix hold-time violations that occur due to the clock skew between two registers, being larger than the data path delay between those same two registers. The FPGA is designed with the clock skew and the clock-to-Q time of the FPGA registers in mind, to ensure that there is no need for a delay cell.

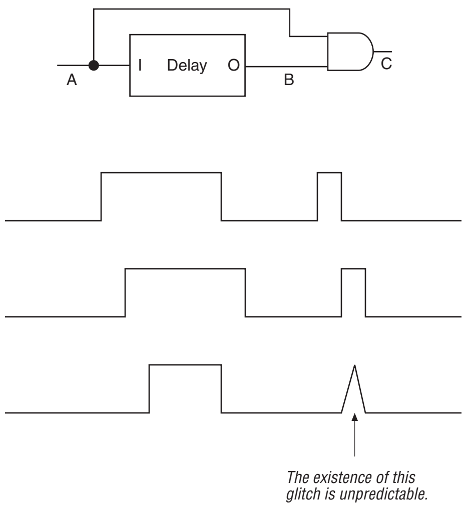
Figure 1–20 shows two versions of the same shift registers. Both circuits operate identically. The first version has a delay cell, possibly implemented using a LUT, in the data path from the Q output of the first register to the D input of the second register. The function of the delay cell is a non-inverting buffer. The second version of this circuit also shows a shift register function, but there is no delay cell in the data path. Both circuits operate identically.

Figure 1–20. Shift Register With and Without an Intentional Delay



If delay chains exist in a design, they are possibly symptomatic of an asynchronous circuit. One such case is shown in the circuit in Figure 1–21. This circuit relies on the delay between two inputs of an AND gate to generate a pulse on the AND gate output. The pulse may or may not be generated, depending on the shape of the waveform on the `A` input pin.

Figure 1–21. A Circuit and Corresponding Timing Diagram Showing a Delay Chain



Using delay chains can cause various design problems, including an increase in a design's sensitivity to operating conditions and a decrease in design reliability.

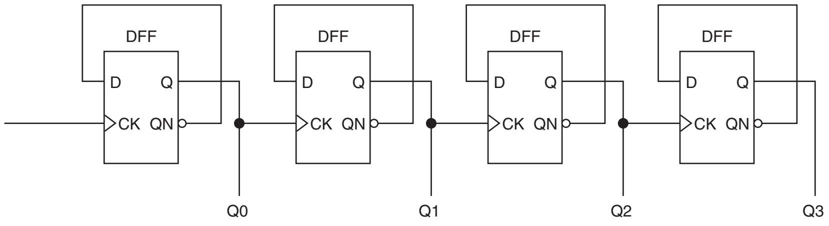
Be aware that not all cases of delay chains in a design are due to asynchronous circuitry. If the Design Assistant report states that you have delay chains that you are unaware of (or are not expecting), the delay chains may be a result of using pre-built intellectual property (IP) functions. Pre-built IP functions may contain delay chains which the Design Assistant reports. These functions are usually parameterizable, and have thousands of different combinations of parameter settings. The synthesis tool may not remove all unused LEs from these functions when particular parameter settings are used, but the resulting circuit is still synchronous. Check all Design Assistant delay chain warnings carefully.

☞ Avoid designing circuits that rely on the use of delay chains, and always carefully check any Design Assistant delay chain warnings.

Ripple Counters

Designs should not contain ripple counters. A ripple counter, shown in Figure 1–22, is a circuit structure where the Q output of the first counter stage drives into the clock input of the following counter stage. Each counter stage consists of a register with the inverted QN output pin feeding back into the D input of the same register.

Figure 1–22. A Typical Ripple Counter



This type of structure is used to make a counter out of the smallest amount of logic possible. However, the LE structure in Altera FPGA devices allows you to construct a counter using one LE per counter-bit, so there is no logic savings in using the ripple counter structure. Each stage of the counter in a ripple counter contributes some phase delay, which is cumulative in successive stages of the counter. Figure 1–23 shows the phase delay of the circuit in Figure 1–22.

Figure 1–23. Timing Diagram Showing Phase Delay of Circuit Shown in Figure 1–22

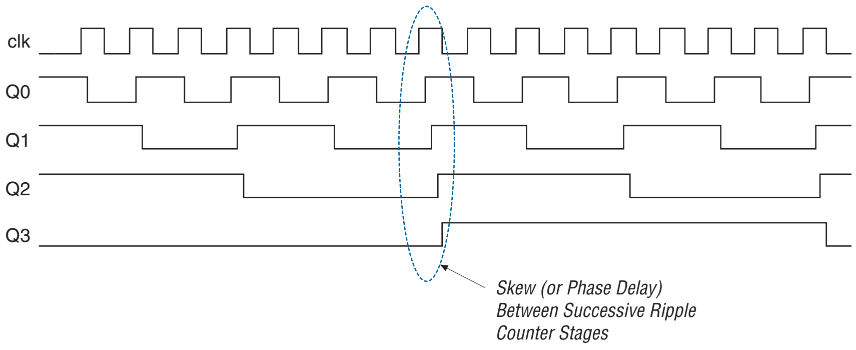
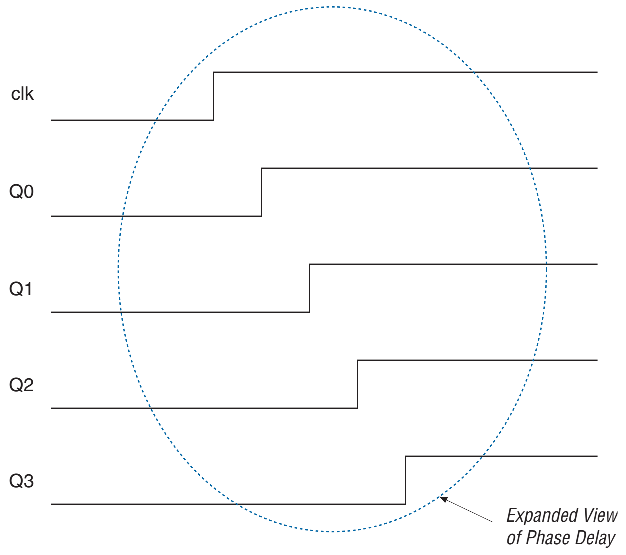


Figure 1–24 shows detailed view of the phase delay shown in Figure 1–23.

Figure 1–24. Detailed View of the Phase Delay Shown in Figure 1–23



This phase delay is problematic if the ripple counter outputs are used as clock signals for other circuits. Those other circuits are clocked by signals that have large skews.

Ripple counters are particularly challenging for static timing analysis tools to analyze as each stage in the ripple counter causes a new clock domain to be defined. The more clock domains that the static timing analysis tool has to deal with, the more complex and time-consuming the process becomes.

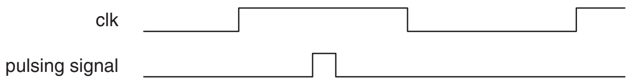
☞ Altera recommends that you avoid using ripple counters under any circumstances.

Pulse Generators

A pulse generator is a circuit that generates a signal that has two or more transitions within a single clock period. Figure 1–25 shows an example of a pulse generator waveform.

☞ For more information on pulse generators, refer to "Intentional Delays" on page 1–18.

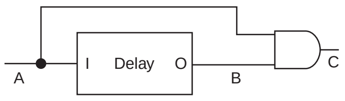
Figure 1–25. Example of a Pulse Generator Waveform



Creating Pulse Generators

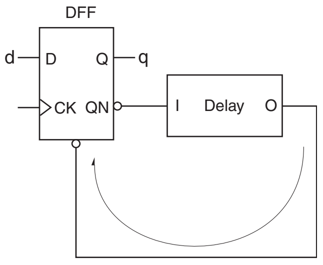
Pulse generators can be created in two ways. The first way to create a pulse generator is to increase the width of a glitch using a 2-input AND, NAND, OR, or NOR gate, where the source for the two gate inputs are the same, but the design delays the source for one of the gate inputs, as shown in Figure 1–26.

Figure 1–26. A Pulse Generator Circuit Using a 2-Input AND



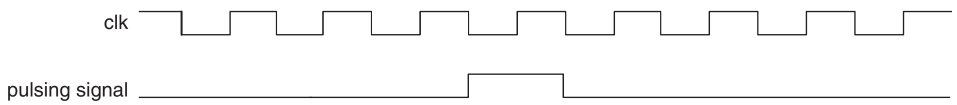
The second way to create a pulse generator is by using a register where the register output drives its own asynchronous reset signal through a delay chain, as shown in Figure 1–27.

Figure 1–27. Pulse Generator Circuit Using a Register Output to Drive a Reset Signal Through a Delay Chain



These pulse generators are asynchronous in nature and are detected by the Design Assistant as unacceptable circuit structures. If you need to generate a pulsed signal, you should do it in a purely synchronous manner. That is, where the duration of the pulse is equal to one or more clock periods, as shown in Figure 1–28.

Figure 1–28. An Example of a Synchronous Pulse Generator



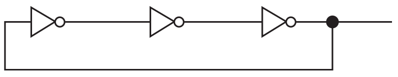
A synchronous pulse generator can be created with a simple section of Verilog HDL or VHDL code. The following is a Verilog HDL code fragment for a synchronous pulse generator circuit.

```
reg [2:0] count;
reg pulse;
always @ (posedge clk or negedge rst)
begin
    if (!rst)
        begin
            count[2:0] <= 3'b000;
            pulse <= 1'b0;
        end
    else
        begin
            count[2:0] <= count[2:0] + 1'b1;
            if (count == 3'b000)
                begin
                    pulse <= 1'b1;
                end
            else
                begin
                    pulse <= 1'b0;
                end
            end
        end
    end
end
```

Combinational Oscillator Circuits

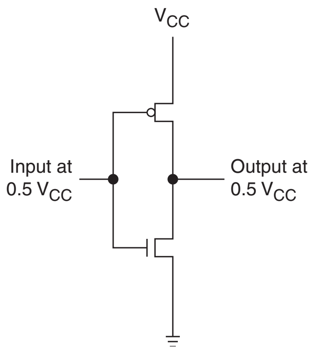
The circuit shown in Figure 1–29 on page 1–24 consists of a combinational logic gate whose inverted output feeds back to one of the inputs of the same gate. This feedback path causes the output to change state and; therefore, oscillate.

Figure 1–29. A Combinational Ring Oscillator Circuit



This circuit is sometimes built out of a series of cascaded inverters in a structure known as a ring oscillator. The frequency at which this circuit oscillates depends on the temperature, voltage, and process operating conditions of the device, and is completely asynchronous to any of the other clock domains in the device. Worse, the circuit may fail to oscillate at all, and the output of the inverter goes to a stable voltage at half of the supply voltage, as shown in Figure 1–30. This causes both the PMOS and NMOS transistors in the inverter chain to be switched on concurrently with a path from V_{CC} to GND, with no inverter function and consuming static current.

Figure 1–30. An Inverter Biased at 0.5 V_{CC}



☞ Avoid implementing any kind of combinational feedback oscillator circuit.

Reset Circuitry

Reset signals are control signals that synchronously or asynchronously affect the state of registers in a design. The special consideration given to clock signals also needs to be given to reset signals. Only the term "reset" is used in this document, but the information described here also applies to "set," "preset," and "clear" signals. Reset signals should only be used to put a circuit into a known initial condition. Also, both the `set` and `reset` pins of the same register should never be used together. If the signals driving them are both activated at the same time, the logic state of the register may be indeterminate.

Gated Reset

A gated reset is generated when combinational logic feeds into the asynchronous reset pin of a register. The gated reset signal may have glitches on it, causing unintentional resetting of the destination register. Figure 1–31 shows a gated reset circuit where the signal driving into the register reset pin has glitches on it causing unintentional resetting.

Figure 1–31. A Gated Reset Circuit and its Associated Timing Diagram

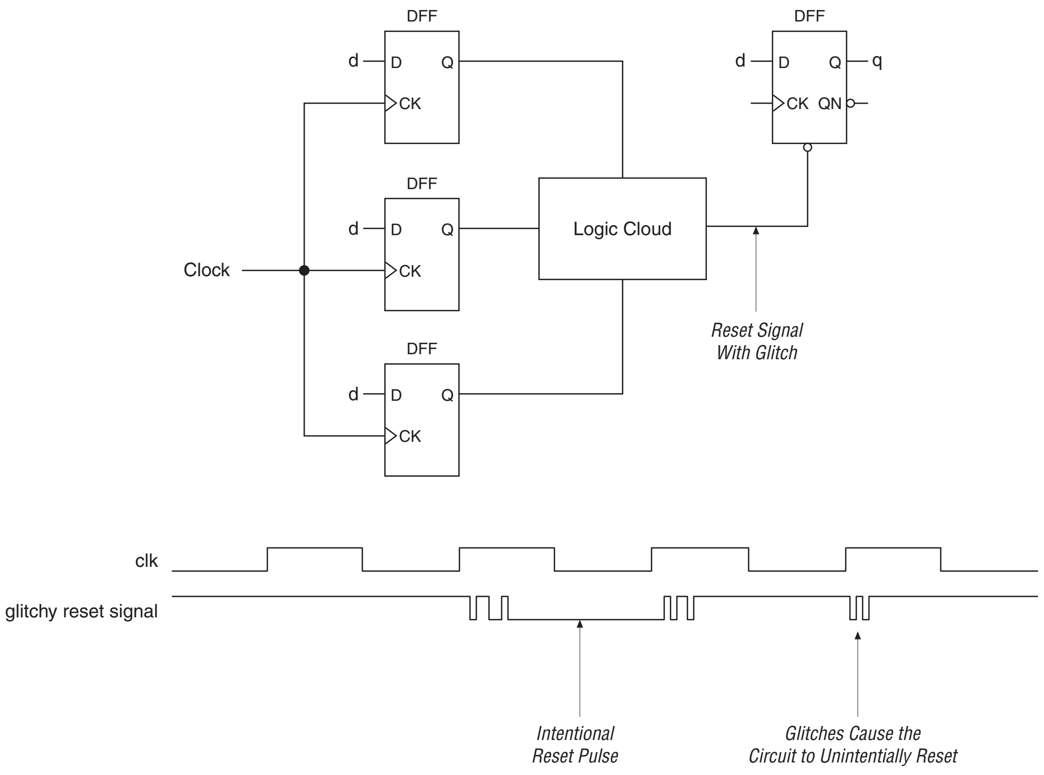
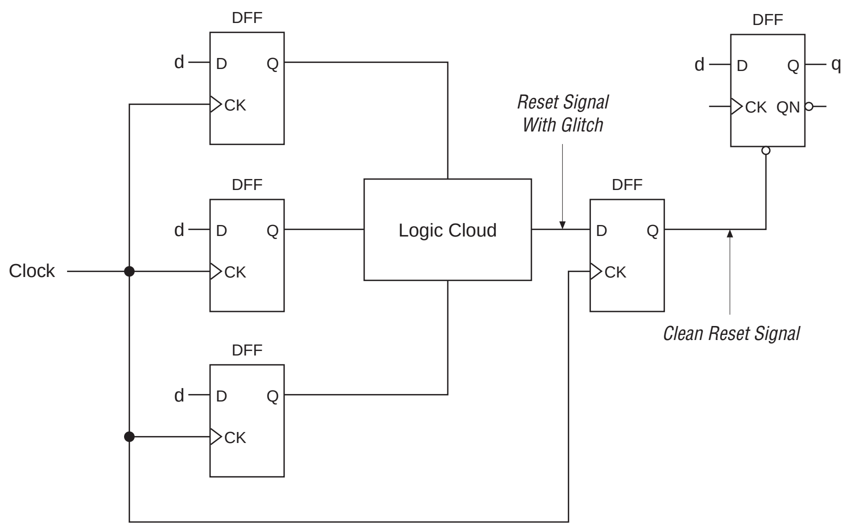


Figure 1–32 shows a better approach to implement a gated reset circuit, by placing a register on the output of the reset-gating logic, thereby synchronizing it to a clock. The register output then becomes a glitch-free reset signal that drives the rest of the design. However, the resulting reset signal is delayed by an extra clock cycle.

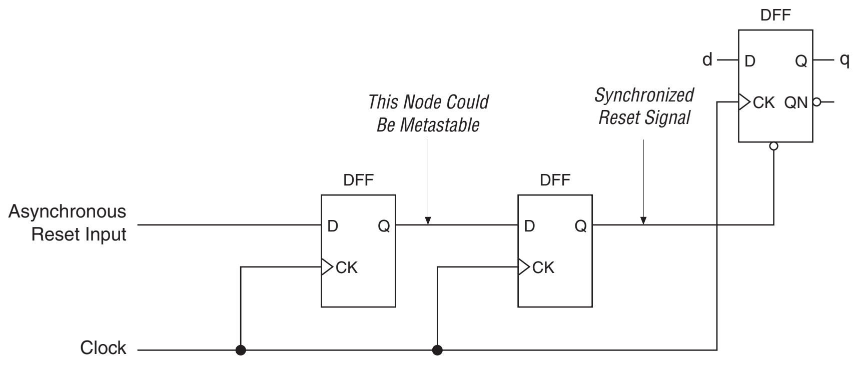
Figure 1–32. A Better Approach to the Gated Reset Circuit in Figure 1–31



Asynchronous Reset Synchronization

If the design needs to be put into a reset state in the absence of a clock signal, the only way to achieve this is through the use of an asynchronous reset. However, it is possible to generate a synchronous reset signal from an asynchronous one by using a double-buffer circuit, as shown in Figure 1–33.

Figure 1–33. A Double-Buffer Circuit



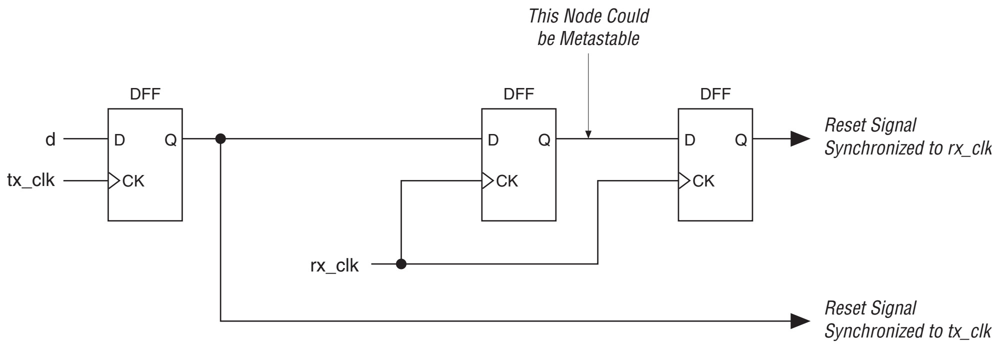
Synchronizing Reset Signals Across Clock Domains

In a design, an internally generated reset signal that is generated in one clock domain, and used in one or more other asynchronous clock domains, should be synchronized. A reset signal that is not synchronized can cause metastability problems.

The synchronization of the gated reset should follow these guidelines, as shown in Figure 1–34.

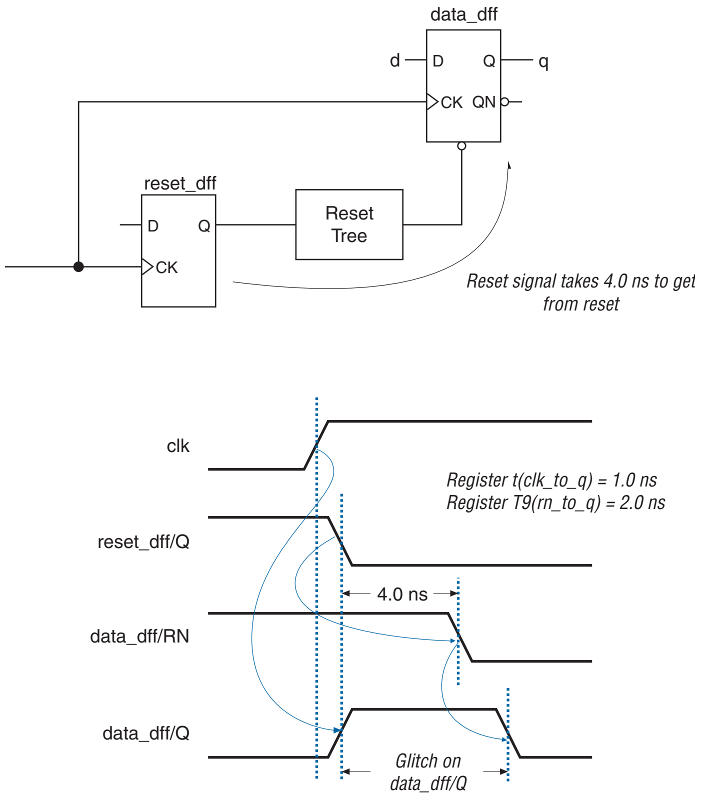
- The reset signal should be synchronized with two or more cascading registers in the receiving asynchronous clock domain.
- The cascading registers should be triggered on the same clock edge.
- There should be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

Figure 1–34. Circuit for a Synchronized Reset Signal Across Two Clock Domains


This Node Could be Metastable
DFF
DFF
DFF
d
tx_clk
D
Q
CK
D
Q
CK
D
Q
CK
rx_clk
Reset Signal Synchronized to rx_clk
Reset Signal Synchronized to tx_clk

With either of the reset synchronization circuits described in Figures 1–33 and 1–34, when the reset is applied, the Q output of the registers in the design may send a wrong signal, momentarily causing some primary output pins to also send wrong signals. The circuit and its associated timing diagram, shown in Figure 1–35, demonstrate this phenomenon.

Figure 1–35. Common Problem with Reset Synchronization Circuits


data_dff
d
D
Q
q
CK
QN
reset_dff
D
Q
CK
Reset Tree
Reset signal takes 4.0 ns to get from reset
clk
reset_dff/Q
data_dff/RN
data_dff/Q
Register t(clk_to_q) = 1.0 ns
Register T9(rn_to_q) = 2.0 ns
4.0 ns
Glitch on data_dff/Q

A purely synchronous reset circuit does not exhibit this behavior. The following Verilog HDL RTL code shows how to do this.

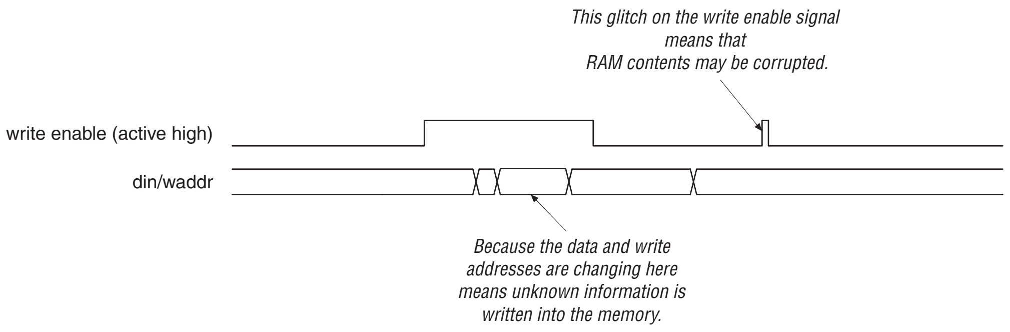
```
always @ (posedge clk)
      begin
      if (!rst)
         q <= 1'b0;
      else
         q <= d; end
```

Avoid using reset signals for anything other than circuit initialization, and be aware of the reset signal timing if reset-synchronizing circuitry is used.

Asynchronous RAM

Altera FPGA devices contain flexible embedded memory structures that can be configured into many different modes. One possible mode is asynchronous RAM. The definition of an asynchronous RAM circuit is one where the write-enable signal driving into the RAM causes data to be written into it, without a clock being required, as shown in Figure 1–36. This means that the RAM is sensitive to corruption if any glitches exist on the write-enable signal. Also, the data and write address ports of the RAM should be stable before the write pulse is asserted, and must remain stable until the write pulse is de-asserted. These limitations in using memory structures in this asynchronous mode imply that synchronous memories are always preferred. Synchronous memories also provide higher design performance.

Figure 1–36. Potential Problems of Using Asynchronous RAM Structures


This glitch on the write enable signal means that RAM contents may be corrupted.
write enable (active high)
din/waddr
Because the data and write addresses are changing here means unknown information is written into the memory.

☞ Stratix, Stratix II, HardCopy Stratix, and HardCopy II device architectures do not support asynchronous RAM behavior. These devices always use synchronous RAM input registers. Altera recommends using RAM output registering; this is optional, however, not using output registering degrades performance.

APEX 20K FPGA and HardCopy APEX support both synchronous and asynchronous RAM using the embedded system block (ESB). Altera recommends using synchronous RAM structures. Immediately registering both input and output RAM interfaces improves performance and timing closure.

Conclusion

Most issues described in this document can be easily avoided while a design is still in its early stages. These issues not only apply to HardCopy devices, but to any digital logic integrated circuit design, whether it is a standard cell ASIC, gate array, or FPGA.

Sometimes, violating one or more of the above guidelines is unavoidable, but understanding the implications of doing so is very important. One must be prepared to justify to Altera the need to break those rules in this case, and to support it with as much documentation as possible.

Following the guidelines outlined in this document can ultimately lead to the design being more robust, quicker to implement, easier to debug, and fitted more easily into the target architecture, increasing the likelihood of success.

Document Revision History

Table 1–2 shows the revision history for this chapter.

Table 1–2. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
September 2008, v3.4	Updated chapter number and metadata.	—
June 2007, v3.3	Minor text edits.	—
December 2006 v3.2	● Added revision history.	Added revision history.
March 2006	Formerly chapter 14; no content change.	—
October 2005, v3.1	● Graphic updates ● Minor edits	—
May 2005, v3.0	Updated the Using a FIFO Buffer section.	—
January 2005, v2.0	● Chapter title changed to *Design Guidelines for HardCopy Series Devices*. ● Updated *Quartus® II Software Supported Versions* ● Updated *HardCopy® Design Center Support* ● Updated heading U*sing a Double Synchronizer for Single-Bit Data Transfer* ● Added *Stratix® II support for a global or regional clock* ● Added *Support for Stratix II and HardCopy II to Mixing Clock Edges*	—

Table 1–2. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
August 2003, v1.1	Edited hierarchy of section headings.	
May 2003, v1.0	Initial release	