

Introduction

Today's programmable logic device (PLD) applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Designs coded optimally will behave in a predictable and reliable manner, even when re-targeted to different device families or speed grades. Good design practices also aid in successful design migration between FPGA and HardCopy™ or ASIC implementations for both prototyping and production. For optimal performance and better time-to-market when designing with Altera® devices, you should understand the impact of synchronous design practices, follow recommended design techniques, follow recommendations for design partitioning, and take advantage of the architectural features in your device.

Synchronous FPGA Design Practices

The first step in a good design methodology is to understand the implications of your design practices and techniques. This section outlines some of the benefits of optimal synchronous design practices and the hazards involved in other techniques. Good synchronous design practices can help you consistently meet your design goals. Inherent problems with other design techniques can include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

The basic principle of synchronous design is that a clock signal triggers all events. As long as all of the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades. In addition, if you plan to migrate your design to a high-volume solution such as Altera HardCopy devices, or if you are prototyping an ASIC, then synchronous design practices help ensure successful migration.



For HardCopy migrations, see the *Design Guidelines for HardCopy Migration* chapter in the *HardCopy Handbook*.

Fundamentals of Synchronous Design

In a synchronous design, everything is related to the clock signal. On every active edge of the clock (usually the rising edge), the data inputs of registers are sampled and transferred to outputs. Following an active clock edge, combinational logic (feeding the data inputs of registers) changes values. This change triggers a period of instability due to propagation delays through the logic as the signals go through a number of transitions and finally settle to new values. Changes happening on data inputs of registers do not affect the values of their outputs until the next active clock edge.

Because the internal circuitry of registers isolates data outputs from inputs, instability in the combinational logic does not affect the operation of the design as long as the following timing requirements are met:

- Before an active clock edge, the data input is settled for at least the setup time of the register.
- After an active clock edge, the data input remains stable for at least the hold time of the register.

When the setup or hold time of a register is violated, the output can be set to an intermediate voltage level between the high and low levels, called a metastable state. Such a state is not fully stable, and small perturbations, like noise in power rails, can return the register to an unpredictable valid state. Various undesirable effects can occur, including increased propagation delays and incorrect output states. In some cases, the output can even oscillate between the two valid states for a relatively long time.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in PLD designs, enabling them to take “short cuts” and save device resources. Asynchronous design techniques have inherent problems such as relying on propagation delays in a device, which leads to incomplete timing constraints and possible glitches and spikes. Because PLDs, especially today's FPGAs, provide large quantities of high-performance logic gates, registers, and memory, resource and performance trade-offs have changed. You should focus on design practices that help you consistently meet your design goals.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. Design structures in PLDs can have varying timing delays, depending on how the design is placed and routed in the device with each compilation. Therefore, it is almost impossible to determine ahead of time the timing delay associated with a particular block of logic. As devices become faster because of process

improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. Specific examples are provided in the next section. Relying on a particular delay also makes asynchronous designs very difficult to migrate to different devices or speed grades, including HardCopy devices or ASICs.

The timing of asynchronous design structures is often difficult or impossible to model with timing assignments and constraints. If you do not have complete or accurate timing constraints, your synthesis and place-and-route tools' timing-driven algorithms may not be able to perform the best optimizations, and reported results will not be complete.

Some asynchronous design structures can generate harmful glitches — pulses that are very short compared with clock periods. Most glitches are generated by combinational logic. When the inputs of combinational logic change, the outputs exhibit a number of glitches before they settle to their new values. Therefore, incorrect values can be propagated through the combinational logic, leading to incorrect values on the outputs in asynchronous designs. In a synchronous design, glitches on the data inputs of registers are normal events that have no negative consequences because the data is not processed until the clock edge.

Recommended Design Techniques

When designing with hardware description language (HDL) code, it is important to understand how a synthesis tool interprets different HDL coding styles and the results to expect. Your coding style can affect logic utilization and timing performance. This section discusses some basic design techniques to ensure optimal synthesis results for Altera designs while avoiding several common causes of unreliability and instability. Design your combinational logic carefully to avoid potential problems, and pay attention to your clocking schemes to maintain synchronous functionality.

Combinational Logic Structures

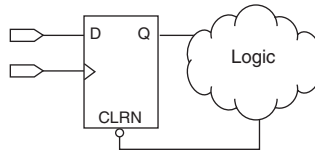
Combinational logic structures are logic functions that depend only on the inputs. In Altera FPGAs, these functions will be implemented in the look-up tables (LUTs) of the device's architecture (logic elements or adaptive logic modules). By following the recommendations in this section, you can help ensure the reliability of your combinational design.

Combinational Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs. In a synchronous design, all feedback loops should include registers. Combinational loops violate synchronous design principles by establishing a direct feedback with no registers. For

example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side. A combinational loop also occurs when you feed back the output of a register to an asynchronous pin of the same register through combinational logic, as shown in [Figure 6-1](#).

Figure 6-1. Combinational Loop through Asynchronous Control Pin



Combinational loops are inherently high-risk design structures for the following reasons:

- Combinational loop behavior generally depends on the relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change and the behavior of the loop may change.
- Combinational loops can cause endless computation loops in many design tools. Most tools break open combinational loops in order to proceed. The various tools used in the design flow may open a given loop in a different manner, processing it in a way that may not be consistent with the original design intent.

Delay Chains

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Often inverters are chained together to add delay. Delay chains generally result from asynchronous design practices, and they are sometimes used to resolve race conditions created by other combinational logic. As discussed above, PLD delays can change with each place-and-route cycle. Delay chains can cause the design problems discussed in the section [“Hazards of Asynchronous Design”](#) on page 6-2.

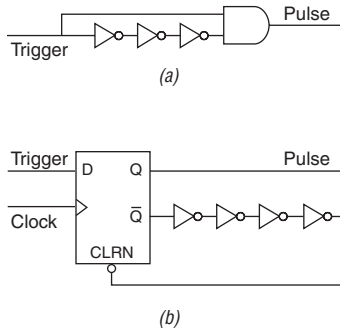
In some ASIC designs, delays may be used for buffering signals as they are routed around the chip. This functionality isn't needed in FPGAs because the routing structure provides buffers throughout the device.

Avoid using delay chains in your design; rely on synchronous practices instead.

Pulse Generators & Multi-Vibrators

You can sometimes use delay chains to generate either one pulse (pulse generators) or a series of pulses (multi-vibrators). There are two common methods for pulse generation as shown in Figure 6–2. These techniques are purely asynchronous and should be avoided.

Figure 6–2. Asynchronous Pulse Generators



In Figure 6–2 (a), a trigger signal feeds both inputs of a 2-input AND or OR gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of inputs. This technique artificially increases the width of the glitch by using a delay chain.

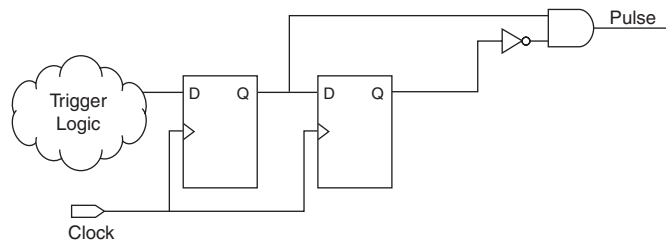
In Figure 6–2 (b), a register's output drives the same register's asynchronous reset signal through a delay chain. The register essentially resets itself asynchronously after a certain delay.

The main problem with these designs is that the pulse widths are difficult for your synthesis and place-and-route software to determine, set, or verify. The actual pulse width can only be determined when routing and propagation delays are known, after placement and routing. So it is difficult to reliably determine the width of the pulse when creating HDL code, and it cannot be set by your electronic design automation (EDA) tools. The pulse may not be wide enough for the application under PVT conditions, and the pulse width changes if you migrate to a different device. In addition, static timing analysis cannot be used to verify the pulse width, so verification is very difficult.

Multi-vibrators use the principle of the “glitch generator” to create pulses, in addition to a combinational loop that turns the circuit into an oscillator. Structures that generate multiple pulses cause even more problems than pulse generators because of the number of pulses involved. In addition, when the structures generate multiples pulses, they also create a new artificial clock in the design.

When you need to use a pulse generator, it should be implemented based on purely synchronous techniques, as shown in [Figure 6–3](#).

Figure 6–3. Recommended Pulse-Generation Technique



In this design, the pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily migrated to other architectures, devices, or speed grades.

Latches

In digital logic, latches hold the value of a signal until a new value is assigned. Latches can also be inferred from HDL code to hold a value when you did not intend to use a latch. FPGAs are register-intensive; therefore, designing with latches uses more logic and leads to lower performance than designing with registers.

Latches can cause various difficulties in the design. Although latches are memory elements like registers, they are fundamentally different. When a latch is in a feed-through or transparent mode, there is a direct path between the data input and the output. Glitches on the data input can pass to the output. The timing for latches is also inherently ambiguous. When analyzing a design with a D latch, for example, the software cannot determine whether you intended to transfer data to the output on the leading edge of the clock or on the trailing edge. In many cases, only the original designer knows the full intent of the design, meaning another designer cannot easily migrate the design or reuse the code.

Clocking Schemes

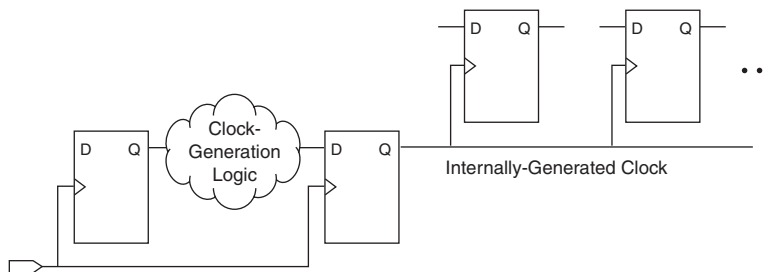
In addition to your combinational logic, your clocking schemes have a large effect on your design's performance and reliability. Avoid using internally generated clocks where possible because they can cause functional and timing problems in the design. Clocks generated with combinational logic can introduce glitches that create functional problems, and the delay due to the combinational logic can lead to timing problems. The following sections provide some specific examples and recommendations to avoid these problems.

Internally-Generated Clocks

If you use the output of combinational logic as a clock signal or as an asynchronous reset signal, you should expect to see glitches in your design. In a synchronous design, glitches on data inputs of registers are normal events that have no consequences. However, a glitch or a spike on the clock input (or on an asynchronous input to a register) can have significant consequences. Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Because of these problems, always register the output of combinational logic before you use it as a clock signal. See [Figure 6-4](#).

Figure 6-4. Recommended Clock-Generation Technique



Registering the output of combinational logic ensures that the glitches generated by the combinational logic are blocked on the data input of the register.

The combinational logic used to generate an internal clock also adds delays on the clock line. In some cases, logic delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register will be violated and the design will not function correctly. To reduce the clock skew within the clock domain, assign the generated clock signal to one of the high-fan-out and low-skew clock networks in the FPGA device (if available). Using a low-skew global clock network such as a global clock signal can help reduce the overall clock skew for the signal. The Quartus® II software will automatically use global routing for high-fan-out control signals. You can make explicit settings using the Assignment Editor when necessary to force the software to use the global routing for particular signals.

Divided Clocks

Many designs require clocks created by dividing a master clock.

Many FPGAs provide dedicated circuitry for clock division, such as Altera's phase-locked loops (PLLs). Using PLL circuitry will avoid many of the problems that can be introduced by asynchronous clock division logic.

When you are using logic to divide a master clock, always use synchronous counters or state machines. In addition, create your design so that registers always directly generate divided clock signals, as described above. Your design should never decode the outputs of a counter or a state machine to generate clock signals; this type of implementation often causes glitches.

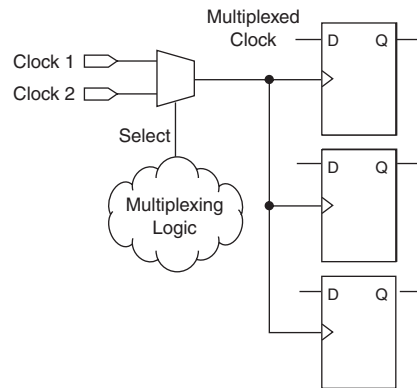
Ripple Counters

FPGA designers in the past had often implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage. This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks have to be handled properly in timing analysis, which can be difficult and may require you to enter appropriate timing assignments in your synthesis and place-and-route tools. To ease verification effort, you should try to avoid these types of structures.

Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. Multiplexing logic of some kind selects a clock source, as in Figure 6-5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 6-5. Multiplexing Logic & Clock Sources



Adding multiplexing logic to the clock signal can lead to some of the problems discussed in the previous sections, but requirements for multiplexed clocks vary widely depending on the application. Clock multiplexing is acceptable if the following criteria are met:

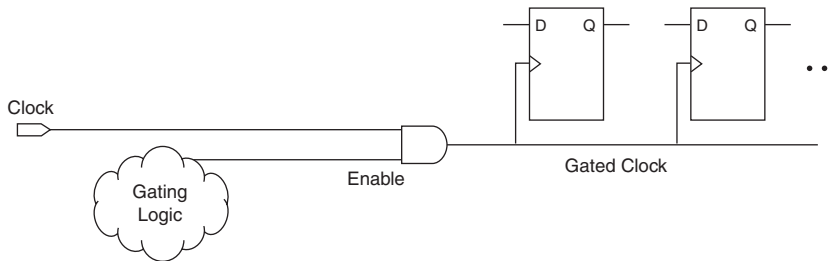
- The clock multiplexing logic does not change after initial configuration
- The design uses multiplexing logic to select a clock for testing purposes
- You always apply a reset when switching clocks
- A temporarily incorrect response of the chip following clock switching has no consequences

If the design switches clocks on the fly with no reset, and your design cannot tolerate a temporarily incorrect response, then you must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry, as in [Figure 6–6](#). When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 6–6. Gated Clock



Gated clocks can be a powerful technique to reduce power consumption. When gating a clock, both the clock network and registers no longer toggle, eliminating their contributions to switching power. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase your design implementation and verification effort, posing challenges in some cases. Gated clocks contribute to clock skew and make device migration difficult. These clocks are also sensitive to glitches, which can cause design failure.

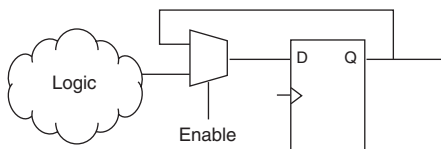
From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable. However, when using a synchronous clock enable scheme, the clock network keeps toggling which does not reduce power consumption as much as gating the clock at the source. Therefore, in most cases, you should use a synchronous scheme outlined in the [“Synchronous Clock Enables”](#) section. However, for major power reduction, see the section [“Recommended Clock-Gating Method”](#) on page 6–11.

Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use synchronous clock enables. Clock enable signals are efficiently supported in most FPGAs with a clock enable signal on the device registers. This scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, but it will perform the

same function as a gated clock by disabling a set of registers. Insert a multiplexer in front of the data input of every register to either load new data or copy the output of the register. See [Figure 6-7](#).

Figure 6-7. Synchronous Clock Enable

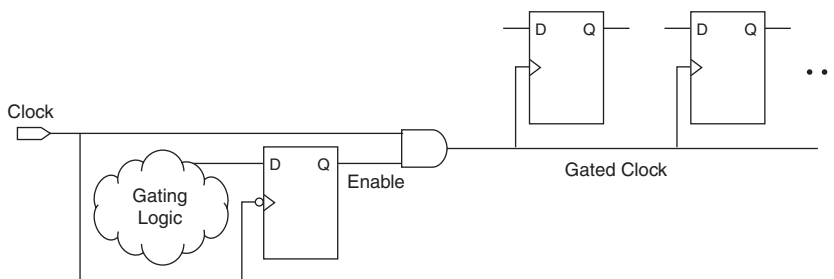


Recommended Clock-Gating Method

Use gated clocks only when your target application requires substantial power reduction. If you must use gated clocks, implement them using the robust clock-gating technique, shown in [Figure 6-8](#).

You can gate a clock signal at the root of the clock network, at each register, or somewhere in between. Because the clock network contributes to switching power, always generate the clock at the root so that you can shut down the entire clock network instead of gating it further along the clock network at the registers.

Figure 6-8. Recommended Clock Gating Technique



As shown in [Figure 6-8](#), a register generates the enable command to ensure that it is free of glitches and spikes. The design clocks the register that generates the enable command on the inactive edge of the clock to be gated (use the falling edge when gating a clock that is active on the rising edge as in [Figure 6-8](#)). With this implementation, only one input of the gate that turns the clock on and off changes at a time, which does not generate glitches or spikes on the output. Use an AND gate to gate a clock

that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable command with a positive edge-triggered register.

When using this scheme, pay attention to the duty cycle of the clock because only the on-time generates the enable command. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, the duty cycle is a minor issue compared with the problems created by other methods of gating clocks.

Hierarchical Design Partitioning

In a hierarchical design, you can create multiple design files and then link them together in a hierarchy. With this technique, you can simulate and optimize the individual modules that comprise the design separately. You can also use the LogicLock design flow to follow a block-based design methodology. When following a hierarchical design methodology, it is important to consider how the design is partitioned. Some synthesis tools have features to help you create separate netlist files or maintain separate parts of a netlist file for different parts of your design, to support block-based design techniques.

Altera recommends the following practices for partitioning designs:

- Partition the design at functional boundaries.
- Minimize the I/O connections between different partitions.
- Do not use “glue logic” between hierarchical blocks. If you preserve hierarchy boundaries, glue logic is not merged with hierarchical blocks. Your synthesis software may optimize glue logic separately, which can degrade synthesis results and is not efficient when used with the LogicLock design methodology.
- Limit clocks to one per block. Partitioning the design into clock domains makes synthesis and timing analysis easier.
- Place state machines in separate blocks to speed optimization and provide greater encoding control.
- Separate timing-critical functions from non-timing-critical functions.
- Limit the critical timing path to one hierarchical block. You can group the logic from several design blocks to ensure the critical path resides in one block.
- Register all inputs and outputs of each block, which makes logic synchronous and avoids glitches. Also, registering outputs may eliminate the need to specify required t_{CO}/t_{SU} or output times for different blocks.

Targeting Clock & Register-Control Architectural Features

In addition to following general design guidelines, it is important to code your design with the target technology in mind. FPGAs provide device-wide clocks and register control signals that can improve performance. Take advantage of your FPGA architecture by following recommended guidelines below.

Clock Network Resources

In ASIC design, an important part of the design process can include balancing the clock delay as it is distributed across the chip. Altera FPGAs provide device-wide global clock routing resources and dedicated inputs, so there is no need to manually balance delays on the clock network. Use the FPGA's low-skew, high-fan-out, dedicated routing where available. By assigning a clock input to one of these dedicated clock pins or making a logic assignment to a global signal, you can take advantage of the dedicated routing available for clock signals.

For best performance, limit the number of global clocks in your design to the number of dedicated global clock resources available in your FPGA. Today's FPGAs offer increasing numbers of global clocks to address large designs with many clock domains. For example, Stratix™ devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are organized into a hierarchical clock structure that allows for up to 22 clocks per device region with low skew and delay, providing up to 48 unique clock domains within the larger Stratix devices. There are 16 dedicated clock pins to drive either the global or regional clock networks. The Stratix enhanced and fast PLL outputs can also drive the global and regional clock networks, and internal signals in the design can be routed onto the clock networks using global logic assignments in the Quartus II software.

To take full advantage of these routing resources, clock signal sources in a design (input clock pins or internally generated clocks) should drive only clock input ports of registers. In certain devices, if a clock signal feeds the data ports of a register, the signal may not be able to use the dedicated routing, which can lead to decreased performance. In general, allowing clock signals to drive the data ports of registers is not considered synchronous design, and it can complicate timing analysis; it is not a recommended practice.

Reset Resources

Take advantage of the device-wide asynchronous reset pin available on most FPGAs. This signal provides low-skew routing across the device. ASIC designs may use local resets to avoid long delays on the signal; however, a global reset signal eliminates these problems.

Register Control Signals

Avoid using an asynchronous load signal if the design's target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture provides only one of those control signals. APEX™ devices, for example, directly support an asynchronous clear function, but not a preset or load function. When the target device does not directly support the signals, the place-and-route software has to use some combinational logic to implement the same functionality. The combinational logic is less efficient and can cause glitches and other problems; it is best to avoid these implementations, if possible.

Conclusion

This chapter discusses the impact of different design methodologies on your quality of results. Fundamental FPGA design guidelines and architecture-specific recommendations are presented. To ensure optimal results in your Altera design, understand the impact of your design practices, follow recommended design techniques for combinational logic and clocking schemes, follow guidelines for design partitioning, and take advantage of architectural features in your FPGA.