Chapter 2
Clocks and Resets

2.1 Introduction

The cost of designing ASICs is increasing every year. In addition to the non-recurring engineering (NRE) and mask costs, development costs are increasing due to ASIC design complexity. To overcome the risk of re-spins, high NRE costs, and to reduce time-to-market delays, it has become very important to design the first time working silicon.

This chapter constitutes a general set of recommendations intended for use by designers while designing a block or an IP (Intellectual Property). The guidelines are independent of any CAD tool or silicon process and are applicable to any ASIC designs and can help designers to plan and to execute a successful System on Chip (SoC) with a well-structured and synthesizable RTL code.

The current paradigm shift towards system level integration (SLI), incorporating multiple complex functional blocks and a variety of memories on a single circuit, gives rise to a new set of design requirements at integration level. The recommendations are principally aimed at the design of the blocks and memory interfaces which are to be integrated into the system-on-chip. However, the guidelines given here are fully consistent with the requirements of system level integration and will significantly ease the integration effort, and ensure that the individual blocks are easily reusable in other systems.

These guidelines can form as a basis of checklist that can be used as a signoff for each design prior to submission for fabrication.

2.2 Synchronous Designs

Synchronous designs are characterized by a single master clock and a single master set/reset driving all sequential elements in the design.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_2,

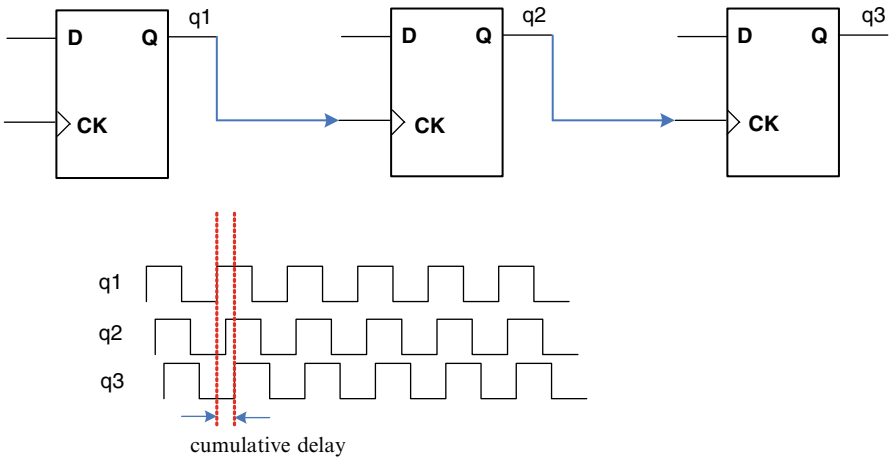

Fig. 2.1 Flip flop driving the clock input of another flip flop (ripple counter)

Experience has shown that the safest methodology for time domain control of an ASIC is synchronous design. Some of the problems with the circuits not being synchronous have been shown in this section.

2.2.1 Avoid Using Ripple Counters

Flip Flops driving the clock input of other flip flops is somewhat problematic. The clock input of the second flip-flop is skewed by the *clock-to-q* delay of the first flip-flop, and is not activated on every clock edge. This cumulative effect with more than two Flip Flops connected in a similar manner forms a Ripple counter as shown in Fig. 2.1. Note the cumulative delay gets added on with more number of flip flops and hence the same is not recommended. More details on the ripple counter are given in Sect. 5.6.7.

2.2.2 Gated Clocks

Gating in a clock line causes clock skew and can introduce spikes which trigger the flip-flop. This is particularly the case when there is a multiplexer in the clock line as shown in Fig. 2.2.

Simulating a gated clock design might work perfectly fine but the problem arises when such a design is synthesized.

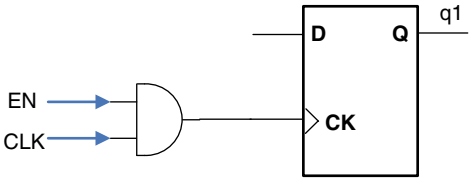

Fig. 2.2 Gated clock line

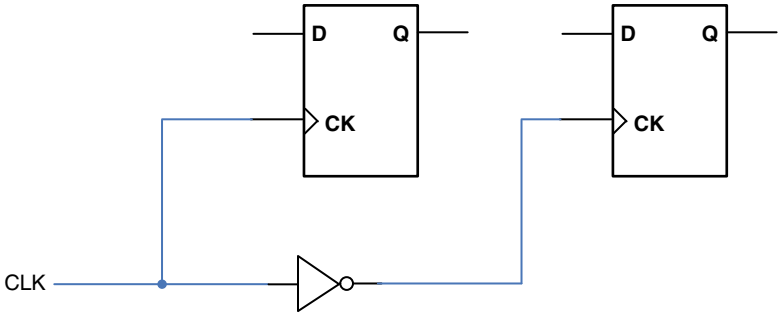

Fig. 2.3 Double-edged clocking

2.2.3 Double-Edged or Mixed Edge Clocking

As shown in Fig. 2.3, the two flip-flops are clocked on opposite edges of the clock signal. This makes synchronous resetting and test methodologies such as scan-path insertion difficult, and causes difficulties in determining critical signal paths.

2.2.4 Flip Flops Driving Asynchronous Reset of Another Flop

In Fig. 2.4, the second flip-flop can change state at a time other than the active clock edge, violating the principle of synchronous design. In addition, this circuit contains a potential race condition between the clock and reset of the second flip-flop.

The subsequent sections show the methods to avoid the above non-recommended circuits.

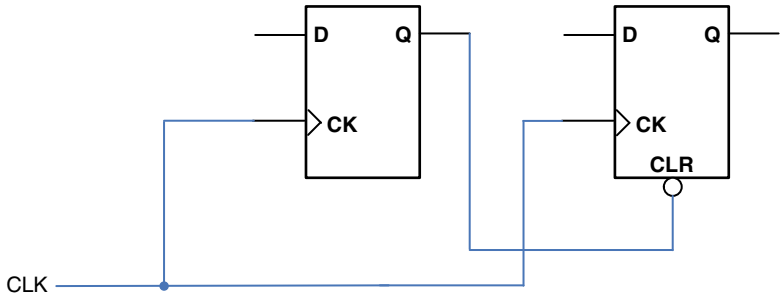

Fig. 2.4 Flip flop driving asynchronous reset of another flop

2.3 Recommended Design Techniques

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL coding styles and the results to expect. It is very important to think in terms of hardware as a particular design style (or rather coding style) can affect gate count and timing performance. This section discusses some of the basic techniques to ensure optimal synthesis results while avoiding several causes of unreliability and instability.

2.3.1 Avoid Combinational Loops in Design

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.

In terms of HDL language, combinational loops occur when the generation of a signal depends on itself through several combinational *always*[1] blocks or when the left-hand side of an arithmetic expression also appears on the right-hand side. Combo loops are a hazard to a design and synthesis tools will always give errors when combo loops are encountered, as these are not synthesize-able.

The generation of combo loops can be understood from the following bubble diagram in Fig. 2.5. Each bubble represents a combo always block and the arrow going into it represents the signal being used in that always block while an arrow going out from the bubble represents the output signal generated by that output block. It is evident that the generation of signal ‘a’ depends on itself through signal ‘d’, thereby generation a combinational loop.

[1]For simplicity, any HDL languages that this book refers to takes Verilog as an example.

```
always@ (a)
begin
b = a;
End

always@ (b)
begin
c = b;
End

always@ (c)
begin
d = c;
End

always@ (c)
Begin
a = c;
end
```

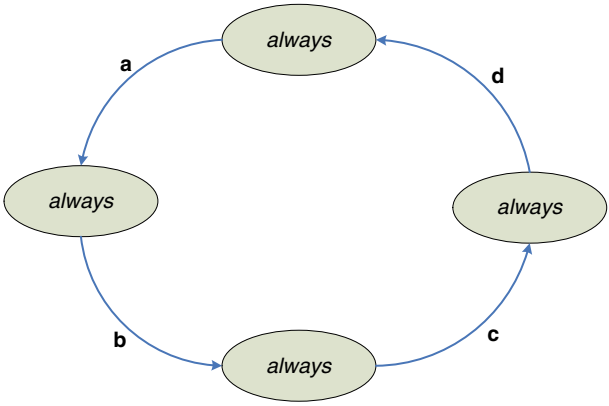

Fig. 2.5 Combinational loop example and bubble diagram

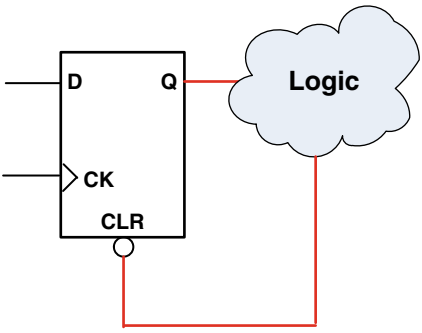

Fig. 2.6 Combinational loop through asynchronous control pins

The code and the bubble diagram are shown below [28]:

In order to remove combo loops, one must change the generation of one of the signals so the dependency of signals on each other is removed. Simple resolution to this problem is to introduce a Flip Flop or register in the combo loop to break this direct path.

Figure 2.6 shows another example where output of a register directly controls the asynchronous pin of the same register through combinational logic.

Combinational loops are inherently high-risk design structures. Combinational loop behavior generally depends on the relative propagation delays through the logic involved in the loop. Propagation delays can change based on various factors and the behavior of the loop may change. Combinational loops can cause endless

computation loops in many design tools. Most synthesis tools break open or disable combinatorial loop paths in order to proceed. The various tools used in the design flow may open a given loop a different manner, processing it in a way that may not be consistent with the original design intent.

2.3.2 Avoid Delay Chains in Digital Logic

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 are sometimes used to resolve race conditions created by other combinational logic. In both FPGA and ASIC, delays can change with each place-and-route. Delay chains can cause various design problems, including an increase in a design's sensitivity to operating conditions, a decrease in a design's reliability, and difficulties when migrating to different device architecture. Avoid using delay chains in a design, rely on synchronous practices instead.

2.3.3 Avoid Using Asynchronous Based Pulse Generator

Often design requires generating a pulse based on some events. Designers 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; these techniques are purely asynchronous and should be avoided:

- A trigger signal feeds both inputs of a two-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 spike by using a delay chain.
- 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.

Asynchronously generated pulse widths often pose problem to the synthesis and place-and-route software. 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. The pulse may not be wide enough for the application in all PVT conditions, and the pulse width will change when migrating to a different technology node. 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 [25].

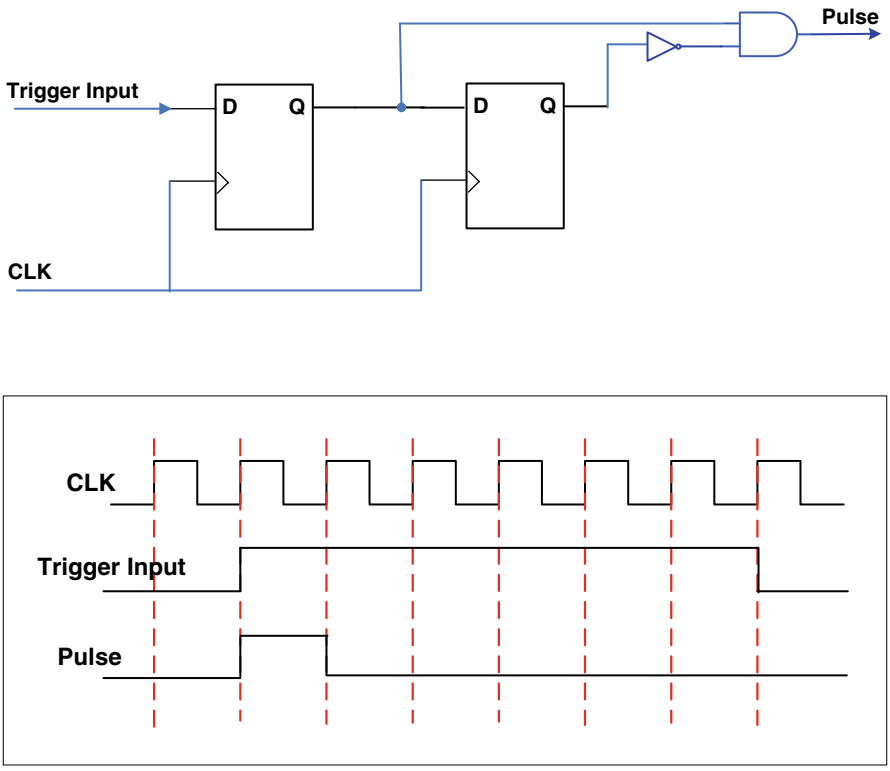

Fig. 2.7 Synchronous pulse generator circuit on start of trigger input

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 increase the frequency of the design.

A recommended Synchronous Pulse generator is shown in Fig. 2.7.

In the above synchronous pulse generator 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 and is technology independent.

Similar to Fig. 2.7, Fig. 2.8 shows the pulse generator at the end of trigger input.

2.3.4 Avoid Using Latches

In digital logic, latches hold the value of a signal until a new value is assigned. Latches should be avoided whereas possible in the design and flip-flops should be used instead.

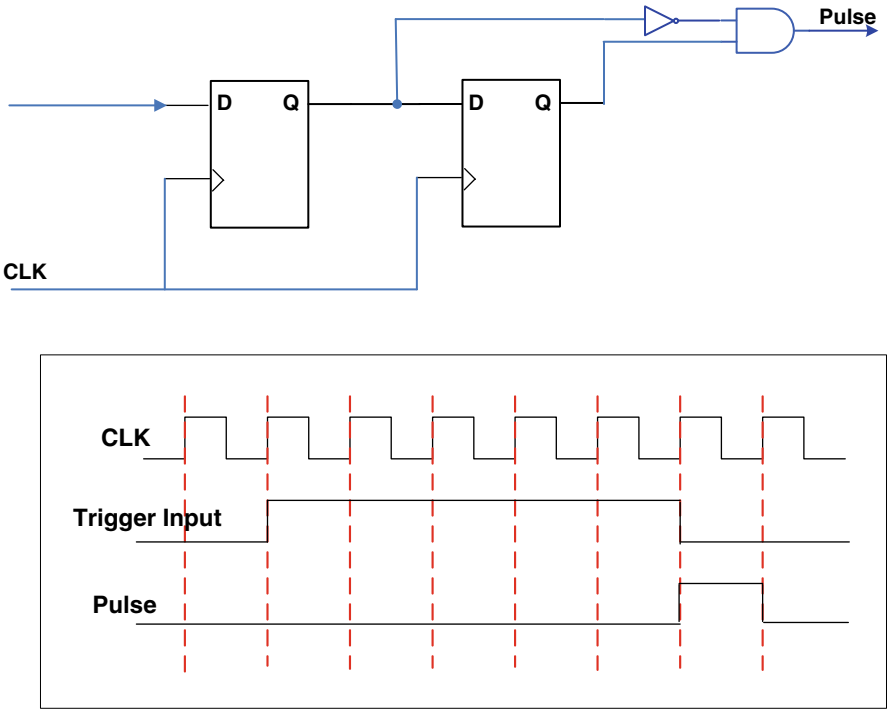

Fig. 2.8 Synchronous pulse generator circuit on end of trigger input

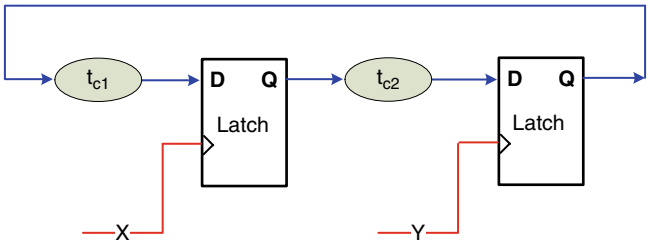

Fig. 2.9 Race conditions in latches

As shown in Fig. 2.9, if both the X and Y were to go high, and since these are level triggered, both the Latches would be enabled resulting in the circuit to oscillate.

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 mode, there is a direct path between the data input and the output. Glitches on the data input can pass to the output.

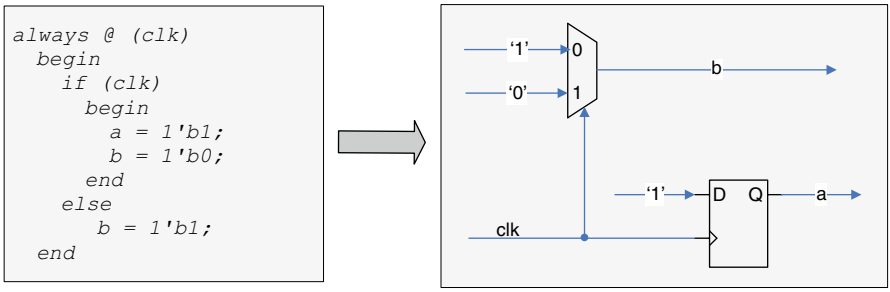

Fig. 2.10 Inferred latch due to incomplete 'if else' statement

Static timing analyzers typically make incorrect assumptions about latch transparency, and either find a false timing path through the input data pin or miss a critical path altogether. The timing for latches is also inherently ambiguous. When analyzing a design with a D latch, for example, the tool 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, which implies that another designer cannot easily migrate the same design or reuse the code.

Latches tend to make circuits less testable. Most design for test (DFT) and automatic test program generator (ATPG) tools do not handle latches very well.

Latches pose different challenge in FPGA designs as FPGA's are register-intensive; therefore, designing with latches uses more logic and leads to lower performance than designing with registers.

Synthesis tools occasionally infer a latch in a design when one is not intended. Inferred latches typically result from incomplete "if" or "case" statements. Omitting the final "else" clause in an "if" or "case" statement can also generate a latch. Figure 2.10 shows a similar example.

As shown in Fig. 2.10, 'b' will be synthesized as straight combinational logic while a latch will be inferred on signal 'a'.

A general rule for latch inferring is that if a variable is not assigned in all possible executions of an always statement (for example, when a variable is not assigned in all branches of an *'if'* statement), then a latch is inferred.

Some FPGA architectures do not support latches. When such a design is synthesized, the synthesis tool creates a combinational feedback loop instead of a latch (as shown in Fig. 2.11).

Combinational feedback loops as shown above are capable of latching data but pose more problem then latches since they may violate setup, hold requirements which are difficult be determined, whereas latches does not have any setup time, hold time violations since they are level triggered.

Note: *The design should not contain any combinational feedback loops. They should be replaced by flip-flops or latches or be eliminated by fully enumerating RTL conditionals.*

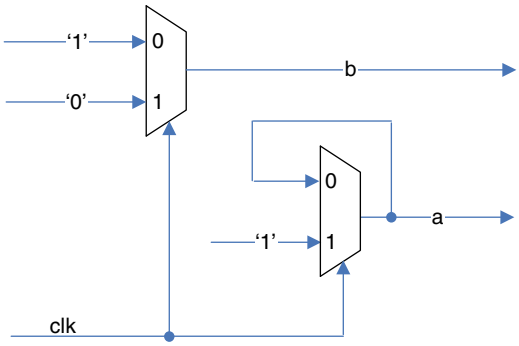

Fig. 2.11 Combinational loop implemented due to incomplete 'if else' statement

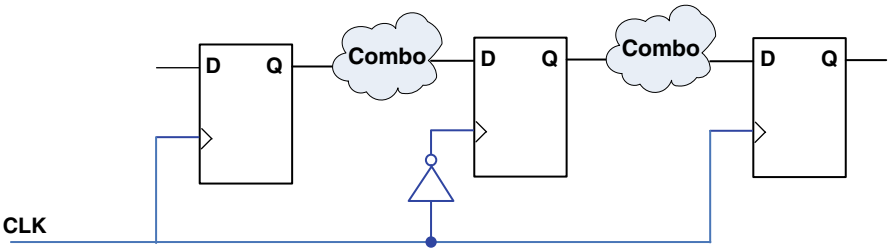

Fig. 2.12 Logic with double edged clocking

To conclude, this does not mean latches should never exist, we will see later how latches could be wonderful when it comes to cycle stealing or time borrowing to meet a critical path in a design.

2.3.5 Avoid Using Double-Edged Clocking

Double or Dual edged clocking is the method of data transfer on both the rising and falling edges of the clock, instead of just one or the other. The change allows for double the data throughput for a given clock speed.

Double edge output stage clocking is a useful way of increasing the maximum possible output speed from a design; however this violates the principle of Synchronous circuits and causes a number of problems.

Figure 2.12 shows a circuit triggered by both edges of clock.

Some of the problems encountered with Double Edged clocking are mentioned below:

- An asymmetrical clock duty cycle can cause setup and hold violations.
- It is difficult to determine critical signal paths.

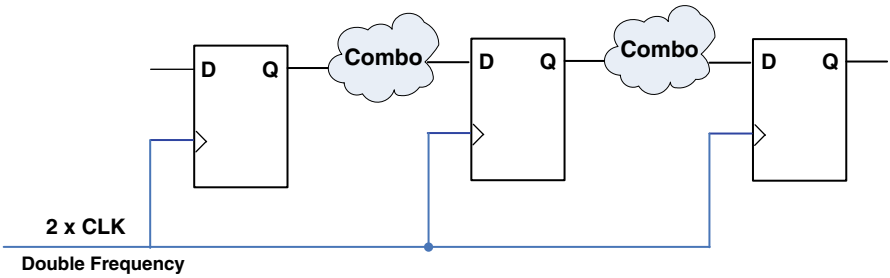

Fig. 2.13 Logic with single edged clocking

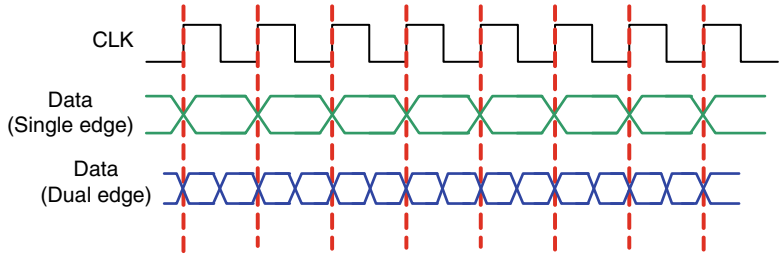

Fig. 2.14 Single/double edged data transfer

- Test methodologies such as scan-path insertion are difficult, as they rely on all flip-flops being activated on the same clock edge. If scan insertion is required in a circuit with double-edged clocking, multiplexers must be inserted in the clock lines to change to single-edged clocking in test mode.

Figure 2.13 shows the normal equivalent pipelined logic with single edge clocking. Note that this synchronous circuit requires a clock frequency that is double the one shown in Fig. 2.12.

Figure 2.14 shows the single transition and double transition clocked data transfer.

The green and blue signals represent data; the "hexagon" shapes are the traditional way of representing a signal that at any given time can be either a one or a zero.

In the circuit shown in Fig. 2.12, an asymmetrical clock duty cycle could cause setup and hold time violations, and a scan-path cannot easily be threaded through the flip-flops.

The above does not means that circuits with dual edge clocking should never be used unless there is an intense desire for higher performance/speed that cannot be met with the equivalent synchronous circuits as the latter comes with an additional overhead of complexity in DFT and verification.

2.3.5.1 Advantages of Dual Edge Clocking

The one constant in the PC world is the desire for increased performance. This in turn means that most interfaces are, over time, modified to allow for faster clocking, which leads to improved throughput. Many newer technologies in the PC world have gone a step beyond just running the clock faster. They have also changed the overall signaling method of the interface or bus, so that data transfer occurs not once per clock cycle, but twice or more.

There are other advantages of circuit operating on dual edge rather than the same synchronous circuit being fed with double the clock frequency. Whatever extent possible, interface designers do regularly increase the speed of the system clock. However, as clock speeds get very high, problems are introduced on many interfaces. Most of these issues are related to the electrical characteristics of the signals themselves. Interference between signals increases with frequency and timing becomes more "tight", increasing cost as the interface circuits must be made more precise to deal with the higher speeds.

The other advantage using double edged clocking is lower power consumptions as clock speeds are decreased by half and hence the system consumes less power than the equivalent synchronous circuits.

So to conclude system integrator should only use dual or double edged clocking unless the same desired performance cannot be met with the equivalent synchronous circuits.

2.4 Clocking Schemes

2.4.1 Internally Generated Clocks

A designer should avoid internally generated clocks, wherever possible, as they can cause functional and timing problems in the design, if not handled properly.

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. In a synchronous design, a glitch on the data inputs does not cause any issues and is automatically avoided as data is always captured on the edge of the clock and thus blocks the glitch. However, a glitch or a spike on the clock input (or an asynchronous input of 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.

Figure 2.15 shows the effect of using a combinational logic to generate a clock on a synchronous counter. As shown in the timing diagram, due to the glitch on the clock edge, the counters increments twice in the clock cycle shown.

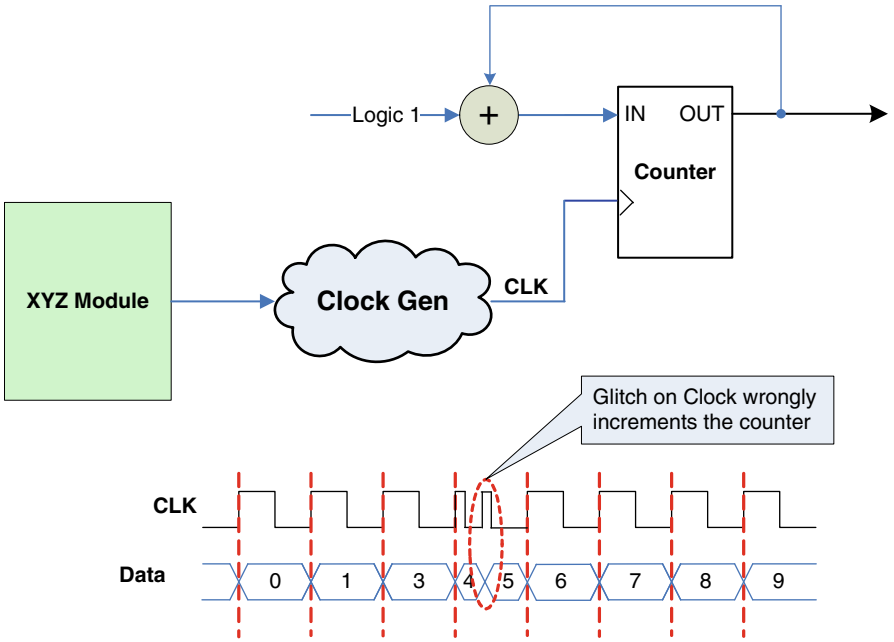

Fig. 2.15 Counter example for using combinational logic as a clock

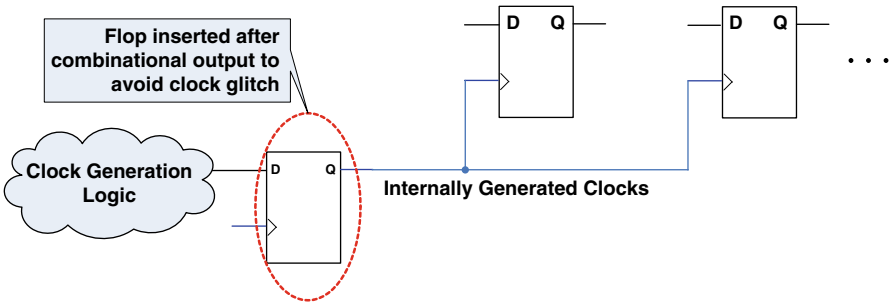

Fig. 2.16 Recommended clock generation technique

This extra counting may create functional issues in the design where instead of counting the desired count, counter counts an additional count due to the glitch on the clock.

Note: *That for the sake of simplicity, it is assumed that the Counters Flops did not violate the setup/hold requirements on the data due to the glitch.*

A simple guideline to the above problem is to always use a registered output of the combinational logic before using it as a clock signal. This registering ensures that the glitches generated by the combinational logic are blocked on the data input of the register (Fig. 2.16).

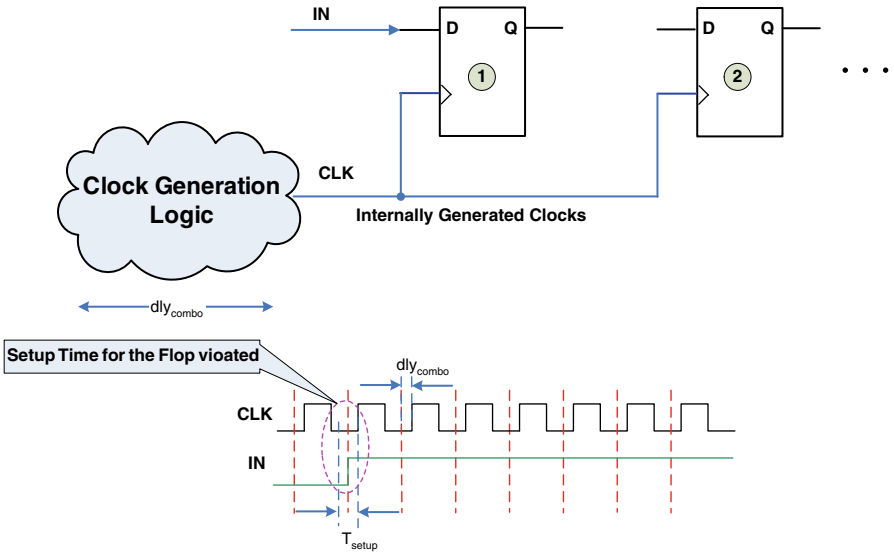

Fig. 2.17 Setup time violated due to skew of clock path

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.

Figure 2.17 shows a similar example where setup time on input "IN" is violated due to skew on the clock path.

Note: *Data path delay is assumed to be zero for simplicity.*

One solution to reduce the clock skew within the clock domain is by assigning the generated clock signal to one of the high-fanout and low-skew clock trees in the SoC. Using a low-skew clock tree can help reduce the overall clock skew for the signal.

2.4.2 Divided Clocks

Many designs require clocks created by dividing a master clock. Design should ensure that most of the clocks should come from the PLL. Using PLL circuitry will avoid many of the problems that can be introduced by asynchronous clock division logic. When using logic to divide a master clock, always use synchronous counters or state machines.

In addition, the design should ensure that registers always directly generate divided clock signals. Design should never decode the outputs of a counter or a state machine to generate clock signals; this type of implementation often causes glitches and spikes.

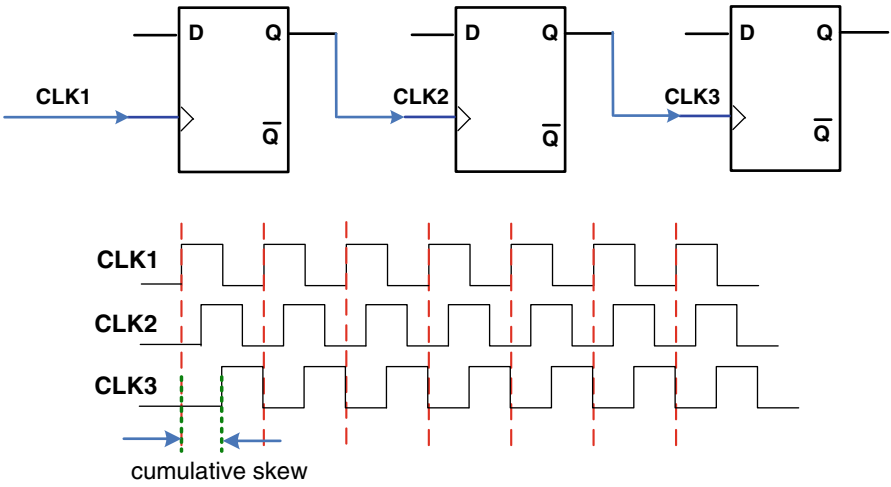

Fig. 2.18 Cascading effort in ripple counters

2.4.3 Ripple Counters

ASIC designers have often implemented ripple counters to divide clocks by a power of 2 because the counters 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 (Fig. 2.18).

This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks pose another set of challenges for STA and synthesis tools. One should try to avoid these types of structures to ease verification effort.

Despite of all the challenges and problems with respect to using Ripple counters, these are quite handy in systems that eat power and can be good to reduce the peak power consumed by a logic or SoC.

Note: *Digital designers should consider using this technique in limited cases and under tight control.*

Refer Chap. 5 "Low power design" on more details analysis and techniques of using Ripple counters to save power consumption.

2.4.4 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 shown in Fig. 2.19.

For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

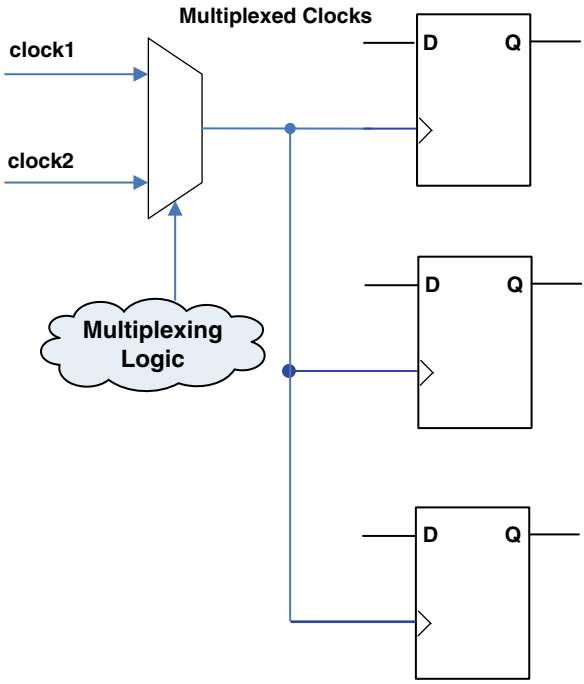

Fig. 2.19 Multiplexing logic and clock sources

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 bypasses functional clock multiplexing logic to select a common clock for testing purposes
- Registers are always in reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks on the fly with no reset and the design cannot tolerate a temporarily incorrect response of the chip, then one 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.

2.4.5 Synchronous Clock Enables and Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry. As shown in Fig. 2.20, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

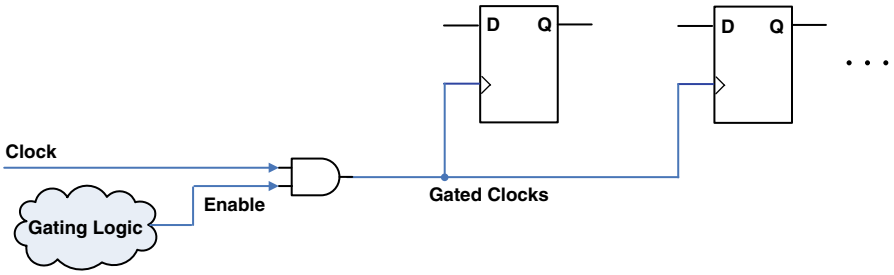

Fig. 2.20 Gated clock

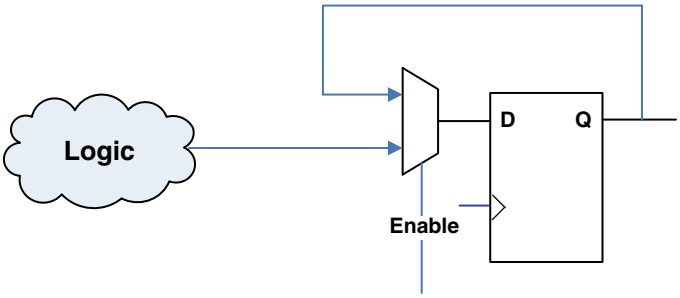

Fig. 2.21 Synchronous clock enable

Gated clocks can be a powerful technique to reduce power consumption. When a clock is gated both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks contribute to clock skew and are also sensitive to glitches, which can cause design failure.

A clock domain can be turned off in a in a purely synchronous manner using a synchronous clock enable. However, when using a synchronous clock enable scheme, the clock tree keeps toggling and the internal circuitry of each Flip Flop remains active (although outputs do not change values), which does not reduce power consumption. A synchronous clock enable technique is shown in Fig. 2.21.

This Synchronous Clock Enable Clocking 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 Flip Flops. As shown in Fig. 2.21, multiplexer in front of the data input of every Flip Flop either load new data or copy the output of the Flip Flop based on the Enable signal.

The next section is dedicated to efficient clock gating methodology that should be used where ever clocking gating is desired due to tight power specifications.

2.5 Clock Gating Methodology

In the traditional synchronous design style, the system clock is connected to the clock pin on every flip-flop in the design. This results in three major components of power consumption:

1. Power consumed by combinatorial logic whose values are changing on each clock edge (due to flops driving those combo cells).
2. Power consumed by flip-flops (this has non-zero value even if the inputs to the flip-flops, and therefore, the internal state of the flip-flops, is not changing).
3. Power consumed by the clock tree buffers in the design.

Gating the clock path substantially reduces the power consumed by a Flip Flop. Clock Gating can be done at the root of the clock tree, at the leaves, or somewhere in between.

Since the clock tree constitutes almost 50% of the whole chip power, it is always a good idea to generate and gate the clock at the root so that entire clock tree can be shut down instead of implementing the gating along the clock tree at the leaves.

Figure 2.22 shows an example of a clock gating for a three bit Counter.

The circuit is similar to the traditional implementation except that a clock gating element has been inserted into the clock network, which causes the flip-flops to be clocked only when the *INC* input is high. When the *INC* input is low, the flip-flops are not clocked and therefore retain the old data. This saves three multiplexers in front of the flip-flops which would had been there in case the gating was implemented by Synchronous Clock Enable as described in Fig. 2.21. This can result in significant area saving when wide banks of registers are being implemented.

2.5.1 Latch Free Clock Gating Circuit

The latch-free clock gating style uses a simple AND or OR gate (depending on the edge on which flip-flops are triggered) as shown in Fig. 2.23.

For the correct operation the circuit imposes a requirement that all enable signals be held constant from the active (rising) edge of the clock until the inactive (falling) edge of the clock to avoid truncating the generated clock pulse prematurely or generating multiple clock pulses (or glitches in clock) where one is required.

Figure 2.24 shows the case where generated clock is truncated prematurely when the above requirement is not satisfied.

This restriction makes the latch-free clock gating style inappropriate for our single-clock flip-flop based design.

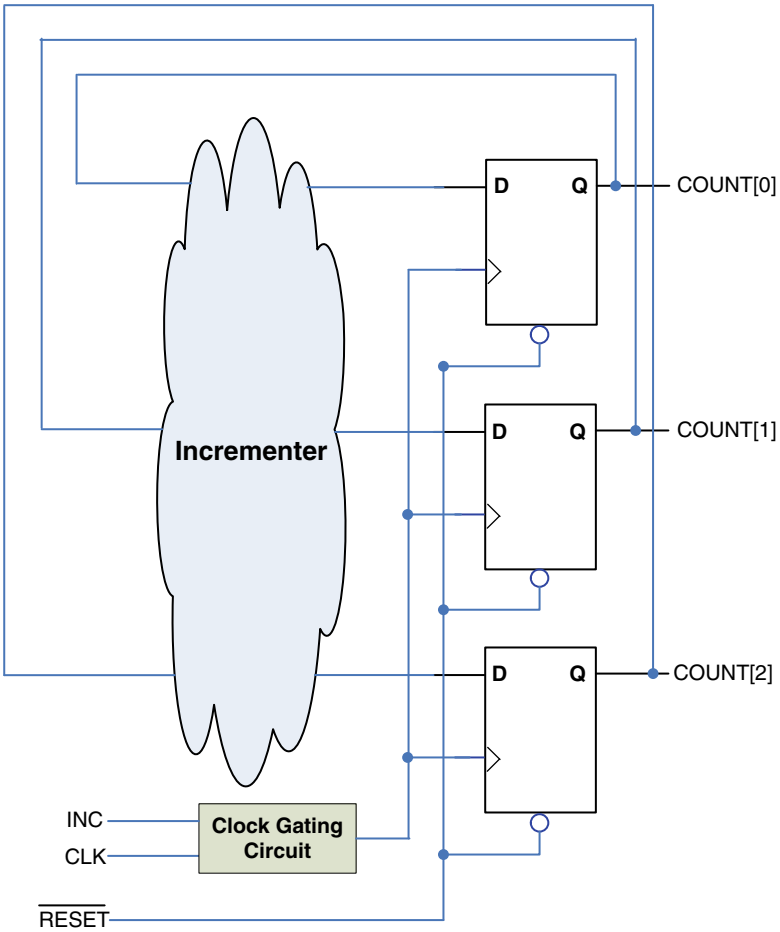

Fig. 2.22 Three bit counter with clock gating

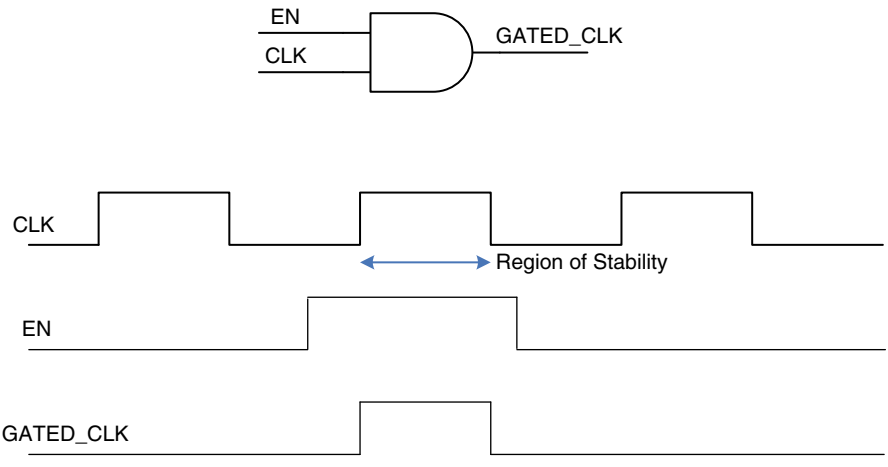

Fig. 2.23 Latch free clock gating circuit

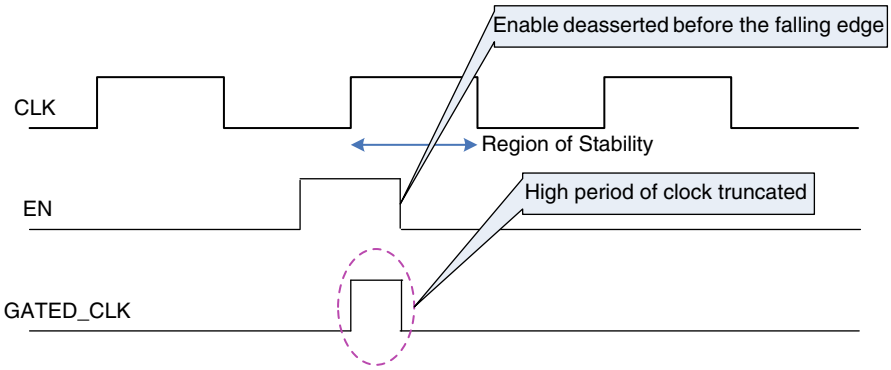

Fig. 2.24 Generated clock terminated prematurely

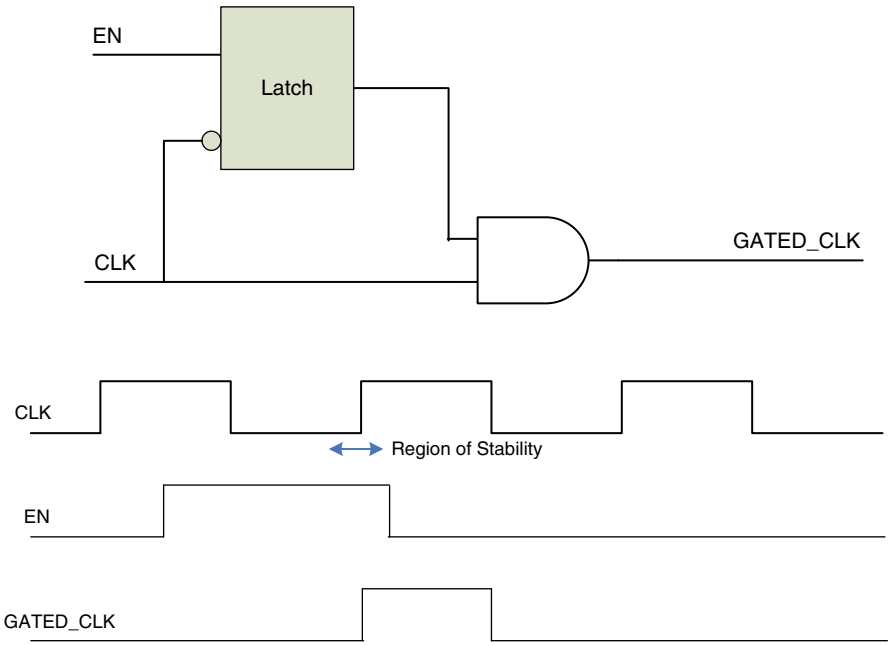

Fig. 2.25 Latch based clock gating circuit

2.5.2 Latch Based Clock Gating Circuit

The latch-based clock gating style adds a level-sensitive latch to the design to hold the enable signal from the active edge of the clock until the inactive edge of the clock, making it unnecessary for the circuit to itself enforce that requirement as shown in Fig. 2.25.

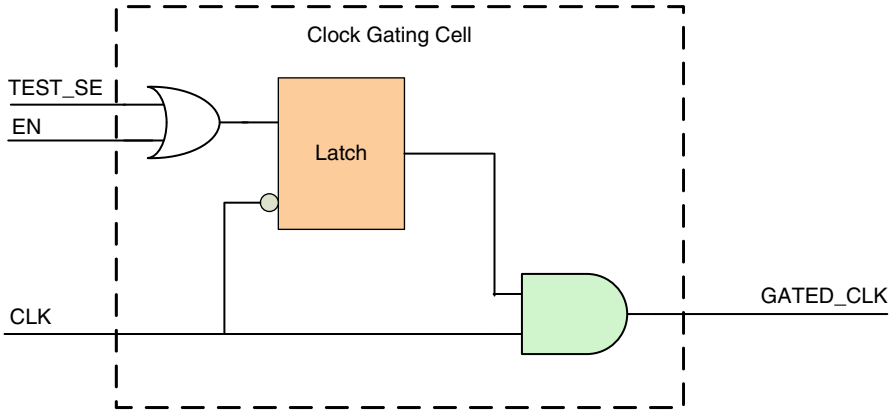

Fig. 2.26 Standard clock gating cell

Since the latch captures the state of the enable signal and holds it until the complete clock pulse has been generated, the enable signal need only be stable around the rising edge of the clock.

Using this technique, only one input of the gate that turns the clock on and off changes at a time, ensuring that the circuit is free from any glitches or spikes on the output.

Note: *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 with a positive edge-triggered Latch.*

When using this technique, special attention should be paid to the duty cycle of the clock and the delay through the logic that generates the enable signal, because the enable signal must be generated in half the clock cycle. 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, being careful with the duty cycle and logic delay may be acceptable compared with the problems created by other methods of gating clocks.

To ensure high manufacturing fault coverage, it is necessary to make sure the clock gating circuit is full controllable and observable to use within a scan methodology. A controllability signal which causes all flip-flops in the design to be clocked, regardless of the enable term value, can be added to allow the scan chain to shift information normally.

This signal can be ORed in with the enable signal before the latch and can be connected to either a test mode enable signal which is asserted throughout scan testing or to a scan enable signal which is asserted only during scan shifting.

The modified circuit is shown in Fig. 2.26. Most of the ASIC vendors do provide this “Clock Gating Cell” as a part of their standard library cell.

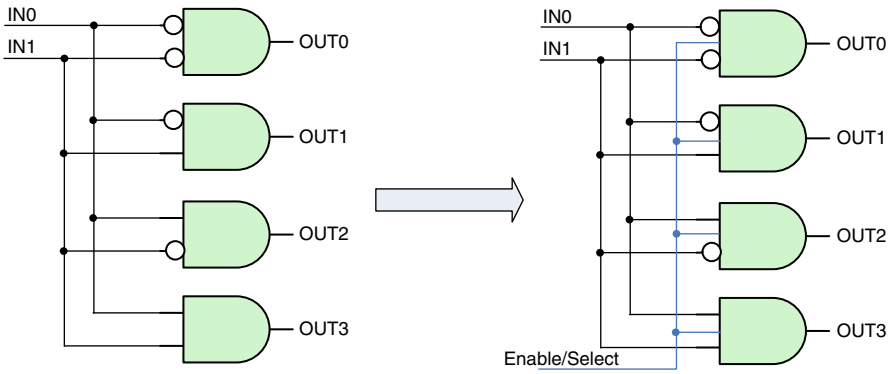

Fig. 2.27 Decoder with enable

2.5.3 Gating Signals

Effective power implementation can be achieved using gating signals for particular parts of the design. Similar to the concept of gating clock, signal gating reduces the transitions in clock free signals. The most common example is the decoder enable.

As part of an address decoding mechanism, signals used by other parts of the design may toggle as a reflection of activity in these parts. Switching activity on one input of the decoder will induce a large number of toggling gates. Controlling this with an enable or select signal prevents the propagation of their switching activity, even if the logic is slightly more complex (Fig. 2.27).

2.5.4 Data Path Re-ordering to Reduce Switching Propagation

Several data path elements, such as decoders or comparison operators, as well as "glitchy" logic may significantly contribute to power dissipation. The glitches, caused by late arrival signals or skews, propagate through other data path elements and logic until they reach a register. This propagation burns more power as the transitions traverse the logic levels. To reduce this wasted dissipation, designers need to rewrite the HDL code and shorten the propagation paths as much as possible. Figure 2.28 illustrates two implementations of the priority mux where the "glitchy" and "stable" conditions are ordered differently.

2.6 Reset Design Strategy

Many design issues must be considered before choosing a reset strategy for an ASIC design, such as whether to use synchronous or asynchronous resets, will every flip-flop receive a reset etc.

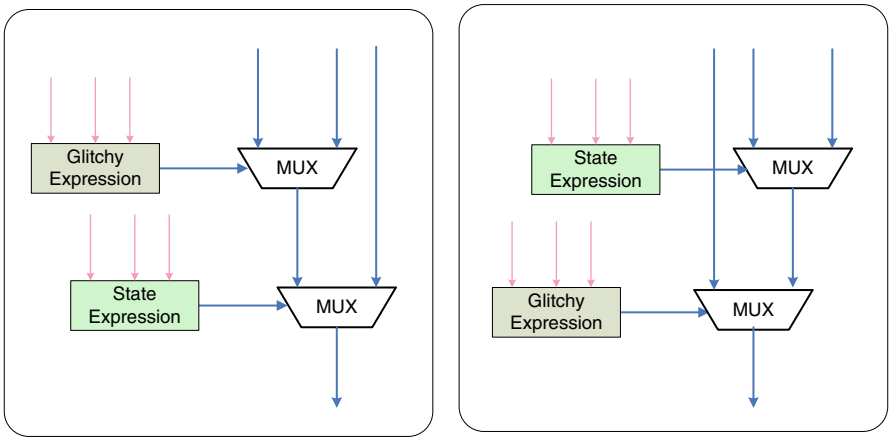

Fig. 2.28 Data path re-ordering to reduce switching propagation

The primary purpose of a reset is to force the SoC into a known state for stable operations. This would avoid the SoC to power on to a random state and get hanged. Once the SoC is built, the need for the SoC to have reset applied is determined by the system, the application of the SoC, and the design of the SoC. A good design guideline is to provide reset to every flip-flop in a SoC whether or not it is required by the system. In some cases, when pipelined flip-flops (shift register flip-flops) are used in high speed applications, reset might be eliminated from some flip-flops to achieve higher performance designs.

A design may choose to use either an Asynchronous or Synchronous reset or a mix of two. There are distinct advantages and disadvantages to use either synchronous or asynchronous resets and either method can be effectively used in actual designs. The designer must use an approach that is most appropriate for the design.

2.6.1 Design with Synchronous Reset

Synchronous resets are based on the premise that the reset signal will only affect or reset the state of the flip-flop on the active edge of a clock. In some simulators, based on the logic equations, the logic can block the reset from reaching the flip-flop. This is only a simulation issue, not a real hardware issue.

The reset could be a "late arriving signal" relative to the clock period, due to the high fanout of the reset tree. Even though the reset will be buffered from a reset buffer tree, it is wise to limit the amount of logic the reset must traverse once it reaches the local logic.

Figure 2.29 shows one of the RTL code for a loadable Flop with Synchronous Reset. Figure 2.30 shows the corresponding hardware implementation.

```
module load_syn_ff ( clk, in, out, load, rst_n);
  input clk, in, load, rst_n;
  output out;

  always @(posedge clk)
    if (!rst_n)
      out <= 1'b0; // sync reset
    else if (load)
      out <= in; // sync

endmodule
```

Fig. 2.29 Verilog RTL code for loadable flop with synchronous reset

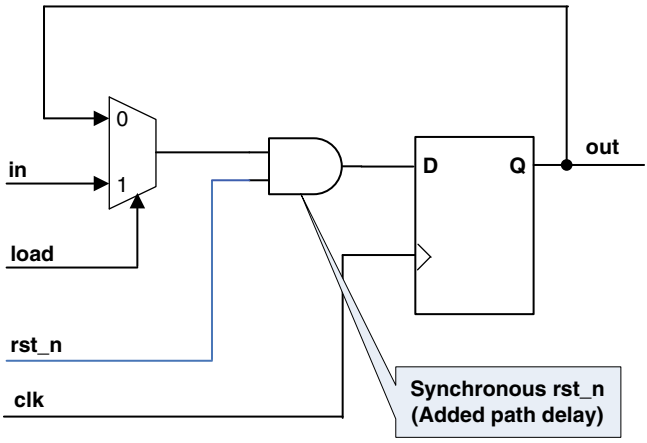

Fig. 2.30 Loadable flop with synchronous reset (hardware implementation)

One problem with synchronous resets is that the synthesis tool cannot easily distinguish the reset signal from any other data signal. The synthesis tool could alternatively have produced the circuit of Fig. 2.31.

Circuit shown in Fig. 2.31 is functionally identical to implementation shown in Fig. 2.30 with the only difference that reset AND gates are outside the MUX. Now, consider what happens at the start of a gate-level simulation. The inputs to both legs of the MUX can be forced to 0 by holding "*rst_n*" asserted low, however if "*load*" is unknown (X) and the MUX model is pessimistic, then the flops will stay unknown (X) rather than being reset. Note this is only a problem during simulation. The actual circuit would work correctly and reset the flop to 0.

Synthesis tools often provide compiler directives which tell the synthesis tool that a given signal is a synchronous reset (or set). The synthesis tool will "pull" this signal as close to the flop as possible to prevent this initialization problem from occurring.

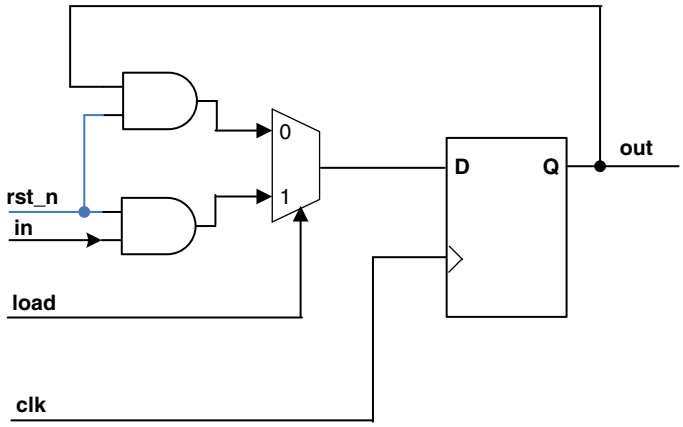

Fig. 2.31 Alternate implementation for loadable flop with synchronous reset

It would be recommended to add these directives to the RTL code from the start of project to avoid re-synthesizing the design late in the project schedule.

2.6.1.1 Advantages of Using Synchronous Resets

1. Synchronous resets generally insure that the circuit is 100% synchronous.
2. Synchronous reset logic will synthesize to smaller flip-flops, particularly if the reset is gated with the logic generating the Flop input.
3. Synchronous resets ensure that reset can only occur at an active clock edge. The clock works as a filter for small reset glitches.
4. In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

2.6.1.2 Disadvantages of Using Synchronous Resets

Not all ASIC libraries have flip-flops with built-in synchronous resets. Since synchronous reset is just another data input, the reset logic can be easily synthesized outside the flop itself (as shown in Figs. 2.30 and 2.31).

1. Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock. This is an issue that is important to consider when doing multi-clock design. A small counter can be used that will guarantee a reset pulse width of a certain number of cycles.
2. A potential problem exists if the reset is generated by combinational logic in the SoC or if the reset must traverse many levels of local combinational logic. During simulation, depending on how the reset is generated or how the reset is applied

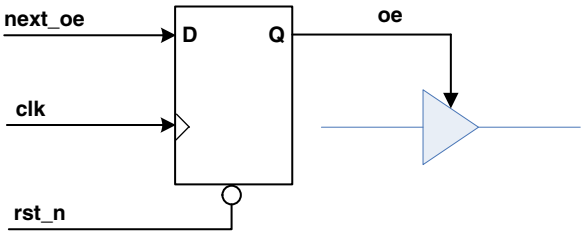

Fig. 2.32 Asynchronous reset for output enable

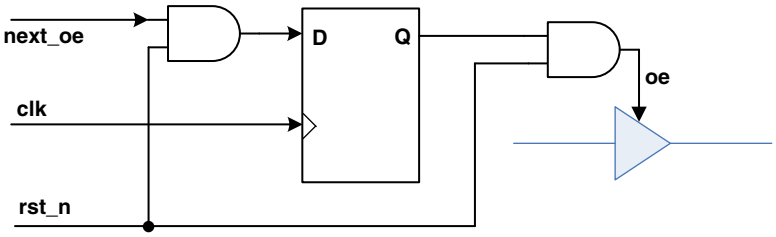

Fig. 2.33 Synchronous reset for output enable

to a functional block, the reset can be masked by X's. The problem is not so much what type of reset you have, but whether the reset signal is easily controlled by an external pin.

3. By its very nature, a synchronous reset will require a clock in order to reset the circuit. This may be a problem in some case where a gated clock is used to save power. Clock will be disabled at the same time during reset is asserted. Only an asynchronous reset will work in this situation, as the reset might be removed prior to the resumption of the clock.

The requirement of a clock to cause the reset condition is significant if the ASIC/FPGA has an internal tristate bus. In order to prevent bus contention on an internal tristate bus when a chip is powered up, the chip should have a power-on asynchronous reset as shown in Fig. 2.32.

A synchronous reset could be used; however you must also directly de-assert the tristate enable using the reset signal (Fig. 2.33). This synchronous technique has the advantage of a simpler timing analysis for the reset-to-HiZ path.

2.6.2 Design with Asynchronous Reset

Asynchronous reset flip-flops incorporate a reset pin into the flip-flop design. With an active low reset (normally used in designs), the flip-flop goes into the reset state when the signal attached to the flip-flop reset pin goes to a logic low level.

```
module load_asyn_ff ( clk, in, out, load, rst_n);
   input clk, in, load, rst_n;
   output out;

   always @(posedge clk or nededge rst_n)
     if (!rst_n)
       out <= 1'b0; // sync reset
     else if (load)
       out <= in; // sync

endmodule
```

Fig. 2.34 Verilog RTL code for loadable flop with asynchronous reset

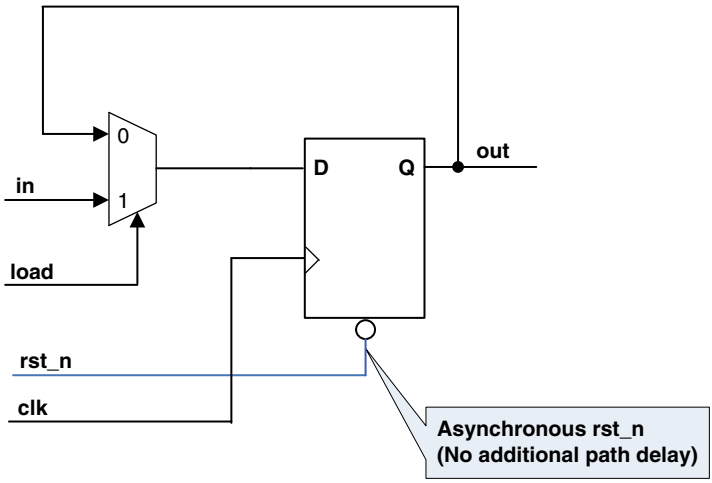

Fig. 2.35 Loadable flop with asynchronous reset (hardware implementation)

Figure 2.34 shows one of the RTL code for a loadable Flop with Asynchronous Reset. Figure 2.35 shows corresponding hardware implementation.

2.6.2.1 Advantages of Using Asynchronous Resets

1. The biggest advantage to using asynchronous resets is that, as long as the vendor library has asynchronously reset-able flip-flops, the data path is guaranteed to be clean. Designs that are pushing the limit for data path timing, cannot afford to have added gates and additional net delays in the data path due to logic inserted to handle synchronous resets. Using an asynchronous reset, the designer is guaranteed not to have the reset added to the data path (Fig. 2.35).

```
module dff_set_reset ( clk, in, out, rst_n, set_n);
  input clk, in, rst_n, set_n;
  output out;

always @(posedge clk or nededge rst_n or negedge set_n)
  if (!rst_n)
    out <= 1'b0; // async reset
  else if (!set_n)
    out <= 1'b1; // async set
  else
    out <= in;
endmodule
```

Fig. 2.36 Verilog RTL for the flop with async reset and async set

2. The most obvious advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present. Synthesis tool tend to infer the asynchronous reset automatically without the need to add any synthesis attributes.

2.6.2.2 Disadvantages of Using Asynchronous Resets

1. For DFT, if the asynchronous reset is not directly driven from an I/O pin, then the reset net from the reset driver must be disabled for DFT scanning and testing [30].
2. The biggest problem with asynchronous resets is that they are asynchronous, both at the assertion and at the de-assertion of the reset. The assertion is a non issue, the de-assertion is the issue. If the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go metastable and thus the reset state of the SoC could be lost.
3. Another problem that an asynchronous reset can have, depending on its source, is spurious resets due to noise or glitches on the board or system reset. Often glitch filters needs to be designed to eliminate the effect of glitches on the reset circuit. If this is a real problem in a system, then one might think that using synchronous resets is the solution.
4. The reset tree must be timed for both synchronous and asynchronous resets to ensure that the release of the reset can occur within one clock period. The timing analysis for a reset tree must be performed after layout to ensure this timing requirement is met. One approach to eliminate this is to use distributed reset synchronizer flip-flop.

2.6.3 Flip Flops with Asynchronous Reset and Asynchronous Set

Most synchronous designs do not have flop-flops that contain both an asynchronous set and asynchronous reset, but at times such a flip-flop is required.

Figure 2.36 shows the Verilog RTL for the Asynchronous Set/Reset Flip Flop.

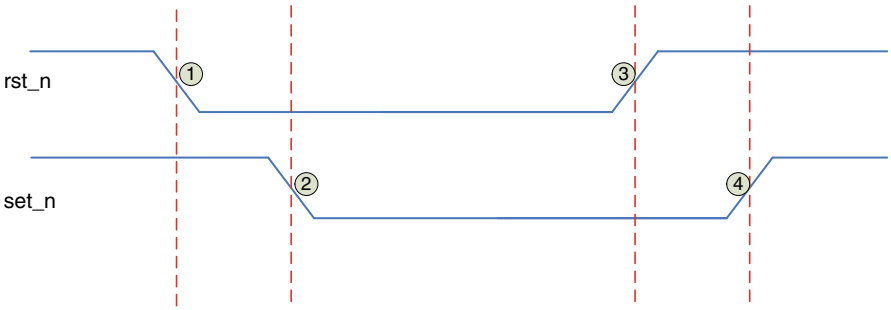

Fig. 2.37 Timing waveform for any asynchronous set/reset condition

```
// Add Compiler specific directive to
// ignore the following block during synthesis
always @(rst_n or set_n)
if (rst_n && !set_n) force q = 1;
else release q;
// End the compiler directive here
endmodule
```

Fig. 2.38 Simulation model for flop with asynchronous set/reset

Synthesis tool should be able to infer the correct flip flop with the asynchronous set/reset but this is not going to work in simulation. The simulation problem is due to the always block that is only entered on the active edge of the set, reset or clock signals.

If the reset becomes active, followed then by the set going active, then if the reset goes inactive, the flip-flop should first go to a reset state, followed by going to a set state (Timing waveform shown in Fig. 2.37).

With both these inputs being asynchronous, the set should be active as soon as the reset is removed, but that will not be the case in Verilog since there is no way to trigger the always block until the next rising clock edge. Always block will be only triggered for 1 and 2 events shown in Fig. 2.37 and would skip the events 3 and 4.

For those rare designs where reset and set are both permitted to be asserted simultaneously and then reset is removed first, the fix to this simulation problem is to model the flip-flop using self-correcting code enclosed within the correct compiler directives and force the output to the correct value for this one condition. The best recommendation here is to avoid, as much as possible, the condition that requires a flip-flop that uses both asynchronous set and asynchronous reset.

The code shown in Fig. 2.38 shows the fix that will simulate correctly and guarantee a match between pre- and post-synthesis simulations.

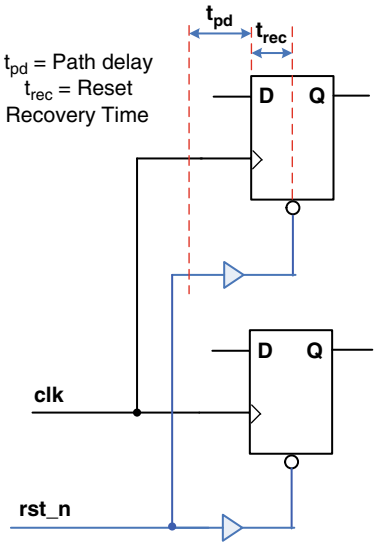

Fig. 2.39 Asynchronous reset removal recovery time problem

2.6.4 *Asynchronous Reset Removal Problem*

Releasing the Asynchronous reset in the system could cause the chip to go into a metastable unknown state, thus avoiding the reset all together. Attention must be paid to the release of the reset so as to prevent the chip from going into a metastable unknown state when reset is released. When a synchronous reset is being used, then both the leading and trailing edges of the reset must be away from the active edge of the clock.

As shown in Fig. 2.39, there are two potential problems when an asynchronous reset signal is de-asserted asynchronous to the clock signal.

1. Violation of reset recovery time. Reset recovery time refers to the time between when reset is de-asserted and the time that the clock signal goes high again. Missing a recovery time can cause signal integrity or metastability problems with the registered data outputs.
2. Reset removal happening in different clock cycles for different sequential elements. When reset removal is asynchronous to the rising clock edge, slight differences in propagation delays in either or both the reset signal and the clock signal can cause some registers or flip-flops to exit the reset state before others.

2.6.5 *Reset Synchronizer*

Solution to asynchronous reset removal problem described in Sect. 2.6.4 is to use a Reset Synchronizer. This is the most commonly used technique to guarantee correct reset removal in the circuits using Asynchronous Resets. Without a reset synchronizer, the usefulness of the asynchronous reset in the final system is void even if the reset works during simulation.

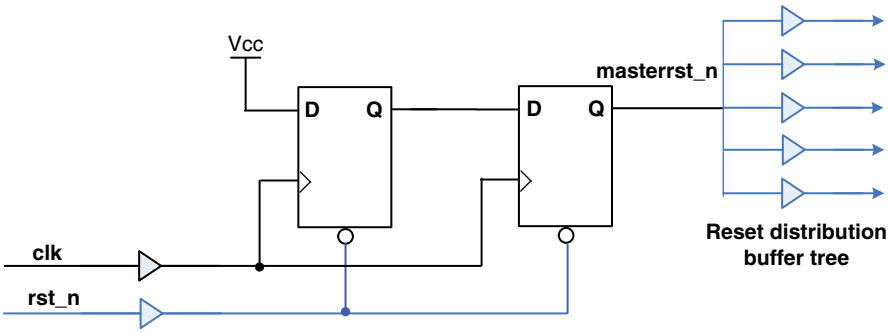

Fig. 2.40 Reset synchronizer block diagram

The reset synchronizer logic of Fig. 2.40 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

An external reset signal asynchronously resets a pair of flip-flops, which in turn drive the master reset signal asynchronously through the reset buffer tree to the rest of the flip-flops in the design. The entire design will be asynchronously reset.

Reset removal is accomplished by de-asserting the reset signal, which then permits the d-input of the first master reset flip-flop (which is tied high) to be clocked through a reset synchronizer. It typically takes two rising clock edges after reset removal to synchronize removal of the master reset.

Two flip-flops are required to synchronize the reset signal to the clock pulse where the second flip-flop is used to remove any metastability that might be caused by the reset signal being removed asynchronously and too close to the rising clock edge.

Also note that there are no metastability problems on the second flip-flop when reset is removed. The first flip-flop of the reset synchronizer does have potential metastability problems because the input is tied high, the output has been asynchronously reset to a 0 and the reset could be removed within the specified reset recovery time of the flip-flop (the reset may go high too close to the rising edge of the clock input to the same flip-flop). This is why the second flip-flop is required.

The second flip-flop of the reset synchronizer is not subjected to recovery time metastability because the input and output of the flip-flop are both low when reset is removed. There is no logic differential between the input and output of the flip-flop so there is no chance that the output would oscillate between two different logic values.

The following equation calculates the total reset distribution time

$$T_{rst\_dis} = t_{clk-q} + t_{pd} + t_{rec}$$

where

t_{clk-q} = Clock to Q propagation delay of the second flip flop in the reset synchronizer
t_{pd} = Total delay through the reset distribution tree
t_{rec} = Recovery time of the destination flip flop

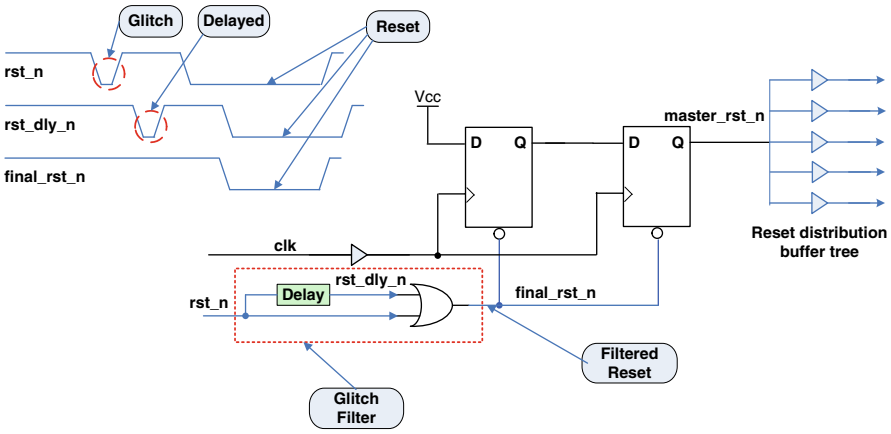

Fig. 2.41 Reset glitch filtering

2.6.6 Reset Glitch Filtering

Asynchronous Reset are susceptible to glitches, that means any input wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. If the reset line is subject to glitches, this can be a real problem. A design may not have a very high frequency sampling clock to detect small glitch on the reset; this section presents an approach that will work to filter out glitches [30]. This solution requires a digital delay to filter out small glitches. The reset input pad should also be a Schmidt triggered pad to help with glitch filtering. Figure 2.41 shows the reset glitch filter circuit and the timing diagram.

In order to add the delay, some vendors provide a delay hard macro that can be hand instantiated. If such a delay macro is not available, the designer could manually instantiate the delay into the synthesized design after optimization. A second approach is to instantiated a slow buffer in a module and then instantiated that module multiple times to get the desired delay. Many variations could expand on this concept.

Since this approach uses delay lines, one of the disadvantages is that this delay would vary with temperature, voltage and process. Care must be taken to make sure that the delay meets the design requirements across all PVT corners.

2.7 Controlling Clock Skew

Difference in clock signal arrival times across the chip is called clock skew. It is a fundamental design principle that timing must satisfy register setup and hold time requirements. Both data propagation delay and clock skew are parts of these

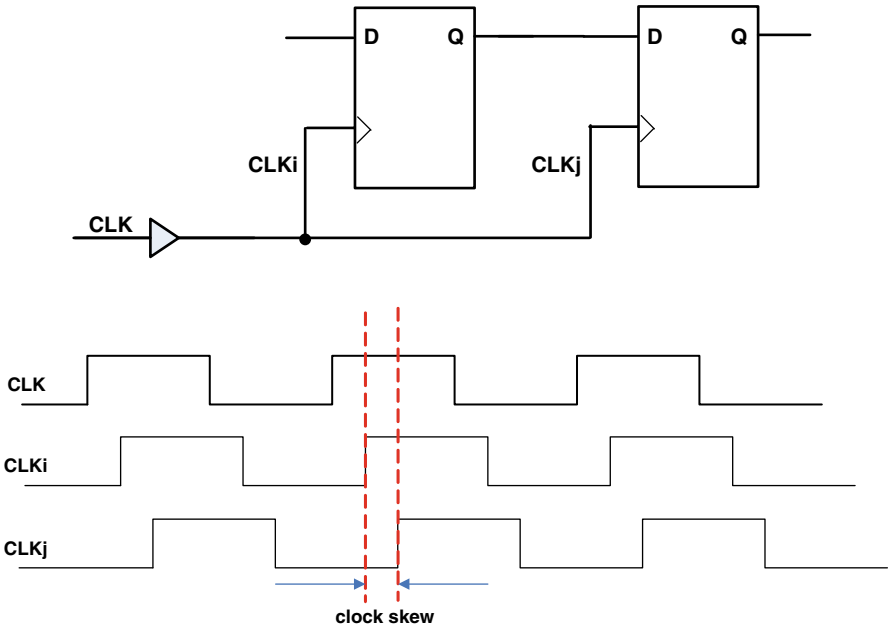

Fig. 2.42 Clock skew in two sequentially adjacent flip-flops

calculations. Clocking sequentially-adjacent registers on the same edge of a high-skew clock can potentially cause timing violations or even functional failures. Probably this is one of the largest sources of design failure in an ASIC.

Figure 2.42 shows an example of clock skew for two sequentially adjacent flip-flops.

Given two sequentially-adjacent flops, F_i and F_j, and an equi-potential clock distribution network, the clock skew between these two flops is defined as

$$\text{Tskew}_{i,j} = T_{c_i} - T_{c_j}$$

where Tc_i and Tc_j are the clock delays from the clock source to the Flops F_i and F_j, respectively.

2.7.1 Short Path Problem

The problem of short data paths in the presence of clock skew is very similar to hold-time violations in flip-flops. The problem arises when the data propagation delay between two adjacent flip-flops is less than the clock skew.

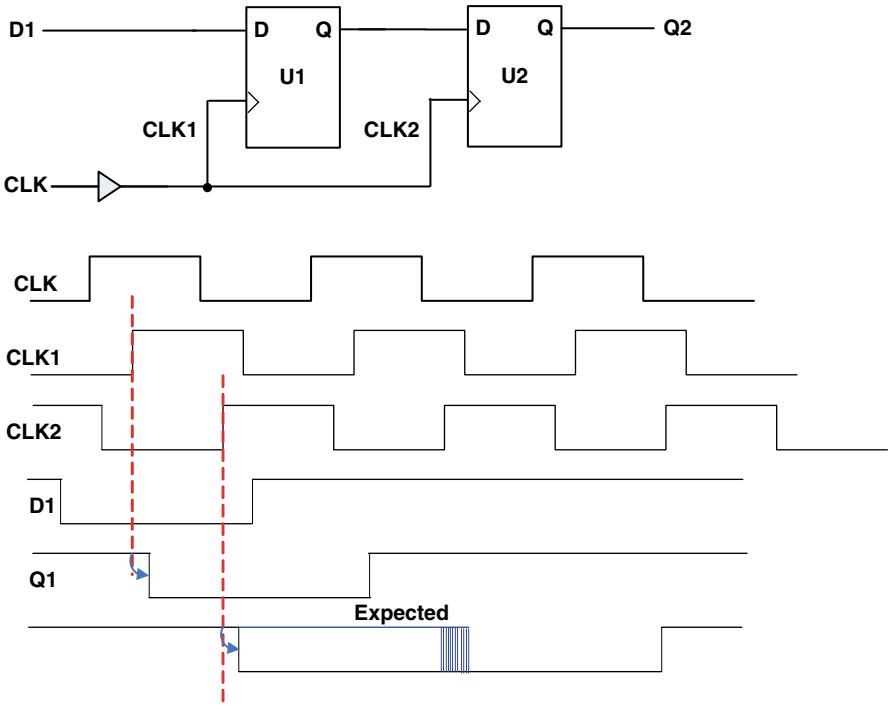

Fig. 2.43 Circuit with a short path problem

Figure 2.43 shows a circuit with timings to illustrate a short-path problem.

Since the same clock edge arrives at the second flip-flop later than the new data, the second flip-flop output switches at the same edge as the first flip-flop and with the same data as the first flip-flop. This will cause U2 to shift the same data on the same edge as U1, resulting in a functional error.

2.7.2 Clock Skew and Short Path Analysis

As mentioned earlier, clock skew and short-path problems emerge when the data propagation path delay between two sequentially adjacent flip-flops is less than the clock skew between the two. Figure 2.44 shows the general diagram of the delay blocks in a sample circuit [33].

The delays in Fig. 2.44 are as follows:

- T_{cq1}: The clock to out delay of the first flip-flop
- T_{rdq1}: The propagation delay from the output of the first flip-flop to the input of the second one
- T_{ck2}: The clock arrival time at the second flip-flop minus the clock arrival time at the first flip-flop

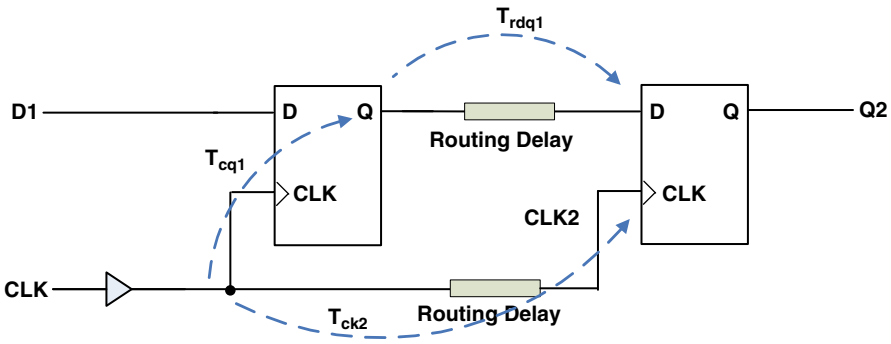

Fig. 2.44 General delay blocks in a simple circuit

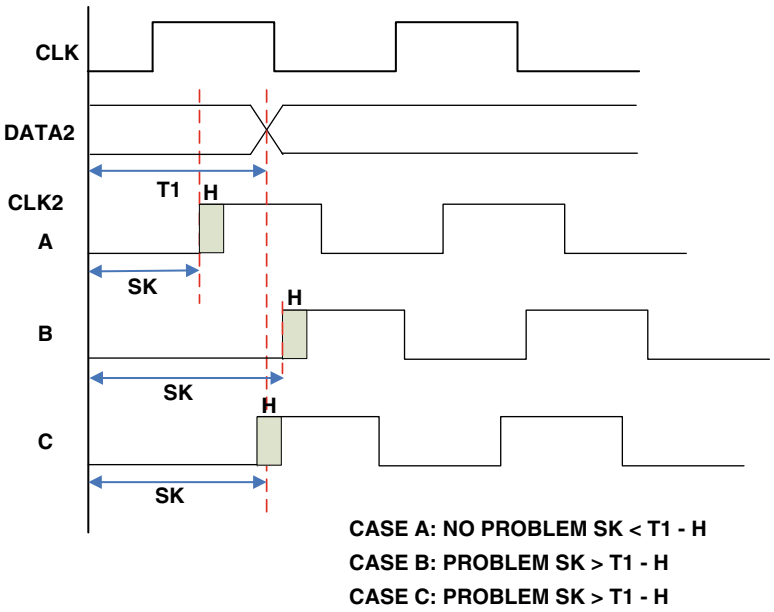

Fig. 2.45 Illustration of short path problem

The short-path problem will definitely emerge in this circuit if

$$T_{ck2} > T_{cq1} + T_{rdq1} - T_{HOLD2}$$

where T_{HOLD2} is the hold-time requirement of the sink flip-flop.

The regions are illustrated in Fig. 2.45.

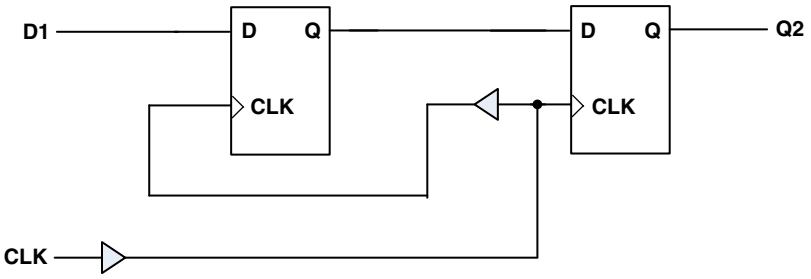

Fig. 2.46 Clock reversing methodology

Therefore, in order to identify the paths with the problem, the user needs to extract the clock skew (e.g. T_{ck2}) and the short-path delays (e.g. $T_{cq1} + T_{rdq1} - T_{HOLD2}$).

2.7.3 Minimizing Clock Skew

Reducing the Clock skew to the minimum is the best approach to reduce the risk of short-path problems. Maintaining the clock skew at a value less than the smallest Flop-to-Flop delay in the design will improve the robustness of the design against any short-path problems.

The following sections are a few well-known design techniques to make designs more robust against clock skew.

2.7.3.1 Adding Delay in Data Path

As Shown in Fig. 2.44, by increasing the Routing Delay in the data path (T_{rdq1}) that eventually increases the total delay of the data path to a value greater than the clock skew, will eliminate the short path problem.

The amount of the inserted delay in the data path should be large enough so that the data path delay becomes sufficiently greater than the clock skew.

2.7.3.2 Clock Reversing

Clock reversing is another approach to get around the problem of short data paths and clock skew. In this technique Clock is applied in the reverse direction with respect to data so that clock skew is automatically eliminated.

The receiving Flop will clock in the transmitting (source) value before the transmitting register receives its clock edge. Figure 2.46 shows a simple example of implementing the clock reversing approach.

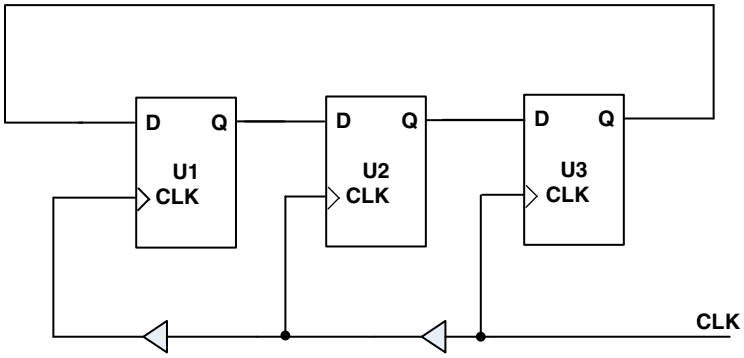

Fig. 2.47 Clock reversing in a circular structure

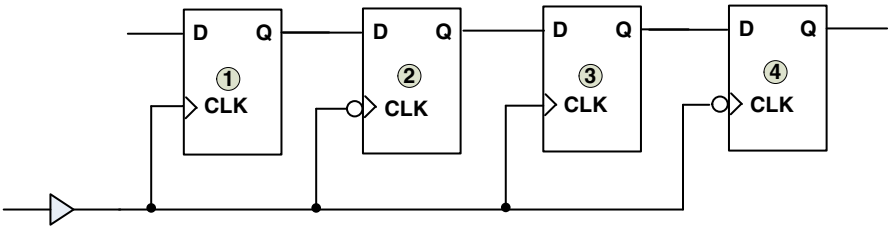

Fig. 2.48 Alternate edge clocking

As shown when sufficient delay is inserted, the receiving Flop will receive the active-clock edge before the source Flop. This improves the Hold time at the expense of Setup Time.

The clock reversing method will not be effective in circular structures such as Johnson counters and Linear Feedback Shift Registers (LFSRs), because it is not possible to define the Sink Flop explicitly. Figure 2.47 shows an example of a circular structure with clock reversing interconnection. As shown, short-path problem exists between flip-flops U1 and U3.

2.7.3.3 Alternate Phase Clocking

One of the known methodologies to avoid clock skew issues is alternate-phase clocking. The following sections mentions few design techniques of alternate phase clocking.

Clocking on Alternate Edges

In this method, sequentially adjacent Flops are clocked on the opposite edges of the clock as shown in Fig. 2.48.

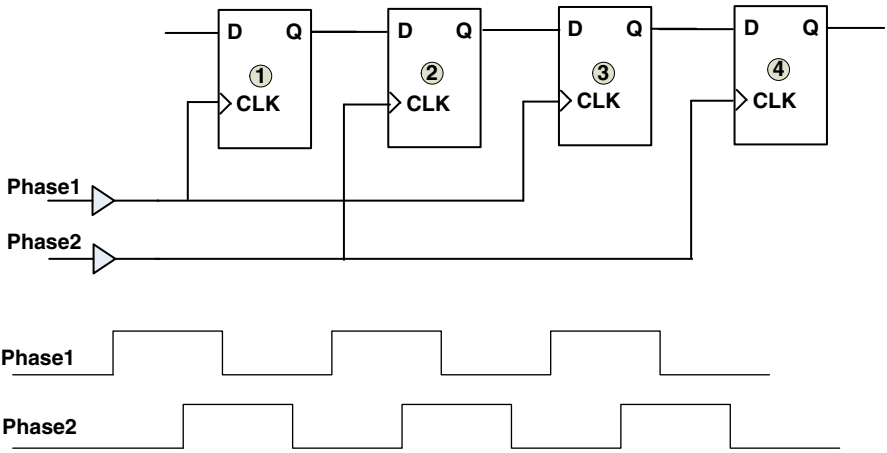

Fig. 2.49 Alternate phase clocking

As shown this method provides a short path-clock skew margin of about one half clock cycle for clock skew.

Clocking on Alternate Phases

Figure 2.49 shows a set of adjacent Flops, which are alternately clocked on two different phases of the same clock. In this case, between each two adjacent Flops, there is a safety margin approximately equal to the phase difference of the two phases.

The user should note that the usage of alternate-phase clocking may require completely different clock constraints on the original clock signal. For example, in the case of clocking on alternate edges, the new constraint on the clock frequency will be half the original frequency since the adjacent Flops are clocked on opposite edges of the same clock cycle.

Ripple Clocking Structure

In a ripple structure, each Flop output drives the next Flop clock port just like the way a Ripple counter is implemented. Here the sink Flop will not clock unless the source Flop toggled as shown in Fig. 2.50.

As shown in Fig. 2.50, the output of each counter flop drives the clock port of the next Flop instead of its data input port. This will eliminate the clock skew since the Flops do not toggle on the same clock. The first Flop is clocked on the positive edge of the CLK signal and the second- and third-stage Flops are clocked on the positive edge of the output of the previous Flop.

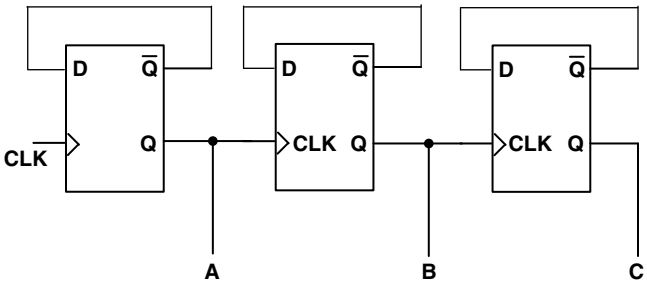

Fig. 2.50 Three bit ripple down-counter

Different techniques as mentioned above may be used to minimize clock skew and avoid short path problems depending on the design complexity and methodology being used.

2.7.3.4 Balancing Trace Length

Techniques described in the previous section are more on design techniques that may be planned much before the final project phase. Of course alternative to the above, Designers may choose to balance the trace length for low skew clock drivers. Apart from merely providing equal traces on all clock nets, the same termination strategy should be used on each trace by placing the same load at the end of the line. This would make sure trace lengths are properly balanced.

Below are some of the guidelines that should be followed:

1. Pay close attention to the specifications for input-to-output delay on the drivers.
2. Use the same drivers at every level of the clock hierarchy.
3. Balance the nominal trace delays at each level.
4. Use the same termination strategy on each line.
5. Balance the loading on each line, even if that means adding dummy capacitors to one branch to balance out loads on the other branches.

References

1. Mohit Arora, Prashant Bhargava, Amit Srivastava, *Optimization and Design Tips for FPGA/ASIC(How to make the best designs)*, DCM Technologies, SNUG India, 2002
2. Application Note, ASIC design guidelines, Atmel Corporation, 1999
3. Cummings CE, Sunburst Design, Inc.; Mills D, LCDM Engineering (2002) Synchronous resets? Asynchronous resets? I am so confused! How will I ever know which to use? SNUG, San Jose
4. Application Note, Clock skew and short paths timing, Actel Corporation, 2004

http://www.springer.com/978-1-4614-0396-8

The Art of Hardware Architecture
Design Methods and Techniques for Digital Circuits
Arora, M.
2012, XV, 221 p., Hardcover
ISBN: 978-1-4614-0396-8