Chapter 10

Reset Circuits

Despite their critical importance, reset circuits are among the most often ignored aspects of an FPGA design. A common misconception among FPGA designers is that reset synchronization is only important in ASIC design and that the global reset resources in the FPGA will handle any synchronization issues. This is simply not true. Most FPGA vendors provide library elements with both synchronous and asynchronous resets and can implement either topology. Reset circuits in FPGAs are critically important because an improperly designed reset can manifest itself as an unrepeatable logical error. As mentioned in previous chapters, the worst kind of error is one that is not repeatable.

This chapter discusses the problems associated with improperly designed resets and how to properly design a reset logic structure. To understand the impact reset has on area, see Chapter 2.

During the course of this chapter, we will discuss the following topics:

- Discussion of asynchronous versus synchronous resets.
  - Problems with fully asynchronous resets
  - Advantages and disadvantages of fully synchronous resets
  - Advantages of asynchronous assertions, synchronous deassertion of reset
- Issues involved with mixing reset types.
  - Flip-flops that are not resettable
  - Dealing with internally generated resets
- Managing resets over multiple clock domains.

10.1 ASYNCHRONOUS VERSUS SYNCHRONOUS

10.1.1 Problems with Fully Asynchronous Resets

A fully asynchronous reset is one that both asserts and deasserts a flip-flop asynchronously. Here, *asynchronous reset* refers to the situation where the reset net is tied to the asynchronous reset pin of the flip-flop. Additionally, the reset assertion and deassertion is performed without any knowledge of the clock. An example circuit is shown in Figure 10.1.

The code for the asynchronous reset of Figure 10.1 is trivial:

```
module resetff(
 output reg  oData,
 input       iClk, iRst,
 input       iData);

always @(posedge iClk or negedge iRst)
 if(!iRst)
   oData <= 0;
 else
   oData <= iData;
endmodule
```

The above coding for a flip-flop is very common but is very dangerous if the module boundary represents the FPGA boundary. Reset controllers are typically interested in the voltage level they are monitoring. During power-up, the reset controller will assert reset until the voltage has reached a certain threshold. At this threshold, the logic is assumed to have enough power to operate in a valid

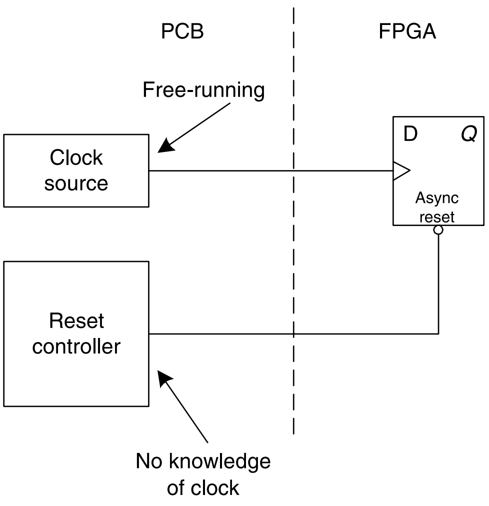

Figure 10.1 Example asynchronous reset source.

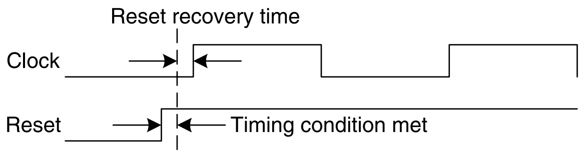

Figure 10.2 Reset recovery time.

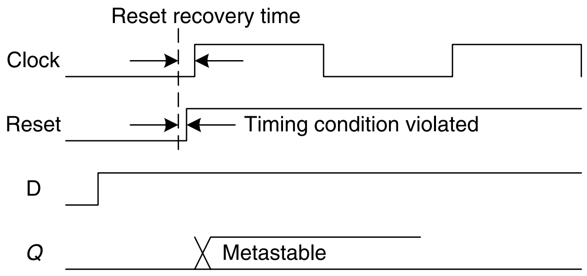

Figure 10.3 Reset recovery time violation.

manner, and so the reset is deasserted. Likewise during a power-down or a brown-out condition, reset is asserted when the voltage rail drops below a corresponding threshold. Again, this is done with no thought to the system clock of the device that is being reset.

The biggest problem with the circuit described above is that it will work most of the time. Periodically, however, the edge of the reset deassertion will be located too close to the next clock edge and violate the reset recovery time. The reset recovery time is a type of setup timing condition on a flip-flop that defines the minimum amount of time between the deassertion of reset and the next rising clock edge as shown in Figure 10.2.

As can be seen in the waveform of Figure 10.2, the reset recovery condition is met when the reset is deasserted with an appropriate margin before the rising edge of the clock. Figure 10.3 illustrates a violation of the reset recovery time that causes metastability at the output and subsequent unpredictable behavior.

Reset recovery time violations occur at the deassertion of reset.

It is important to note that reset recovery time violations only occur on the deassertion of reset and not the assertion. Therefore, fully asynchronous resets are not recommended. The solutions provided later in this chapter regarding the reset recovery compliance will be focused on the transition from a reset state to a functional state.

10.1.2 Fully Synchronized Resets

The most obvious solution to the problem introduced in the preceding section is to fully synchronize the reset signal as you would any asynchronous signal. This is illustrated in Figure 10.4.

The code to implement a fully synchronous reset is similar to the double-flopping technique for asynchronous data signals.

```
module resetsync(
  output reg  oRstSync,
  input       iClk, iRst);
  reg         R1;

always @(posedge iClk) begin
  R1         <= iRst;
  oRstSync   <= R1;
 end
endmodule
```

The advantage to this type of topology is that the reset presented to all functional flip-flops is fully synchronous to the clock and will always meet the reset recovery time conditions assuming the proper buffering is provided for the high fan-out of the synchronized reset signal. The interesting thing about this reset topology is actually not the deassertion of reset for recovery time compliance as discussed in the previous section but rather the assertion (or more specifically the duration of reset). In the previous section, it was noted that the assertion of reset is not of interest, but that is true only for asynchronous resets and not necessarily with synchronous resets. Consider the scenario illustrated in Figure 10.5.

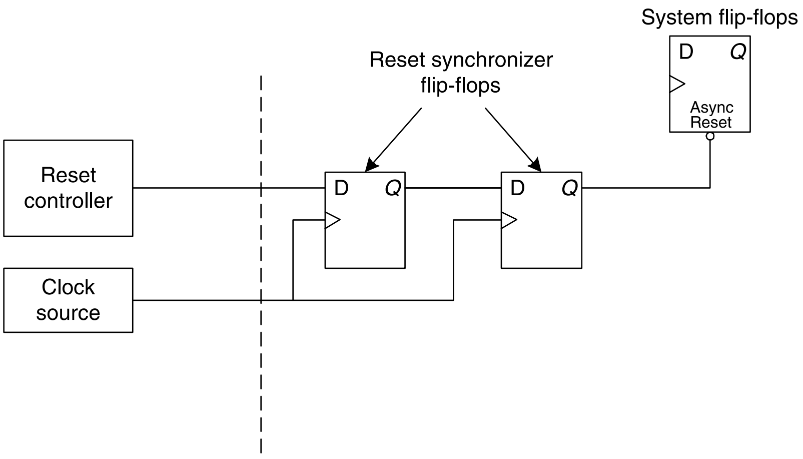

Figure 10.4 Fully synchronized reset.

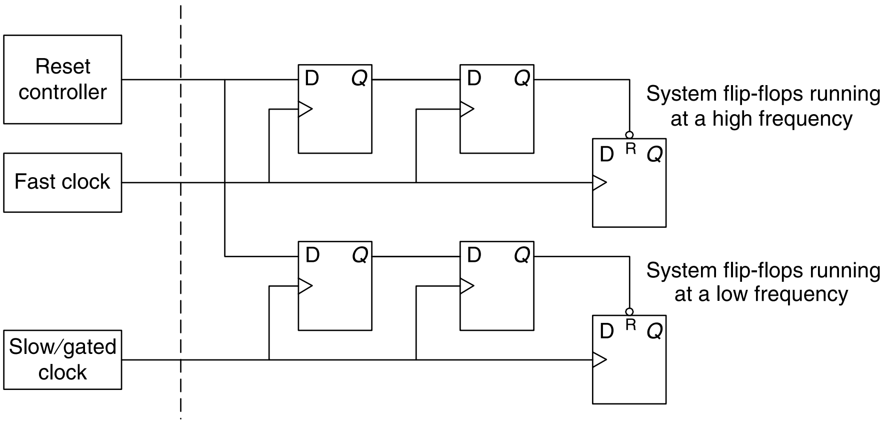
Reset controller
Fast clock
Slow/gated clock
D Q
D Q
D R Q
D Q
D Q
D R Q
System flip-flops running at a high frequency
System flip-flops running at a low frequency

Figure 10.5 Fully synchronous reset with slow/gated clock.

Here, a single reset is synchronized to a fast clock domain and a relatively slow clock domain. For purposes of illustration, this could also be a non-periodic-enabled clock. What happens to the circuit's ability to capture the reset under conditions where the clock is not running? Consider the waveforms shown in Figure 10.6.

In the scenario where the clock is running sufficiently slow (or when the clock is gated off), the reset is not captured due to the absence of a rising clock edge during the assertion of the reset signal. The result is that the flip-flops within this domain are never reset.

Fully synchronous resets may fail to capture the reset signal itself (failure of assertion) depending on the nature of the clock.

For this reason, fully synchronous resets are not recommended unless the capture of the reset signal (reset assertion) can be guaranteed by design. Combining the last two reset types into a hybrid solution that asserts the reset asynchronously and deasserts the reset synchronously would provide the most reliable solution. This is discussed in the next section.

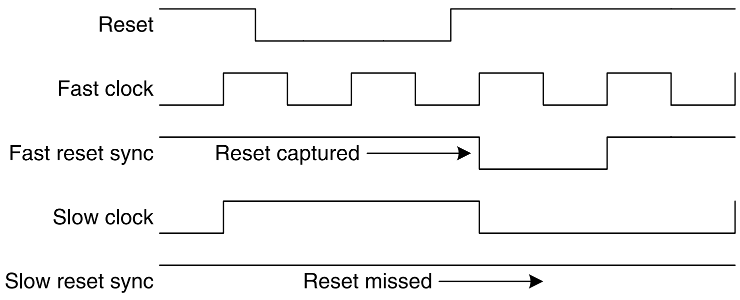

Figure 10.6 Reset synchronization failure.

10.1.3 Asynchronous Assertion, Synchronous Deassertion

A third approach that captures the best of both techniques is a method that asserts all resets asynchronously but deasserts them synchronously.

In Figure 10.7, the registers in the reset circuit are asynchronously reset via the external signal, and all functional registers are reset at the same time. This occurs asynchronous with the clock, which does not need to be running at the time of the reset. When the external reset deasserts, the clock local to that domain must toggle twice before the functional registers are taken out of reset. Note that the functional registers are taken out of reset only when the clock begins to toggle and is done so synchronously.

A reset circuit that asserts asynchronously and deasserts synchronously generally provides a more reliable reset than fully synchronous or fully asynchronous resets.

The code for this synchronizer is shown below.

```
module resetsync(
  output reg  oRstSync,
  input       iClk, iRst);
  reg         R1;

always @(posedge iClk or negedge iRst)
  if(!iRst) begin
    R1        <= 0;
    oRstSync  <= 0;
  end
```

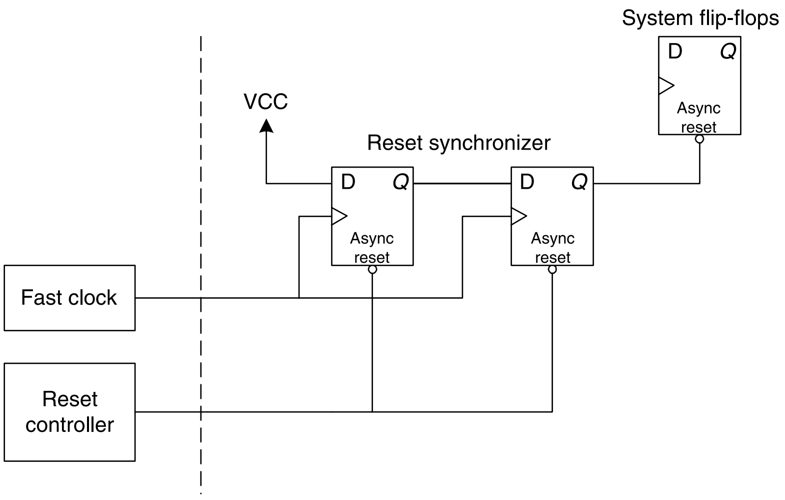

Figure 10.7 Asynchronous assertion, synchronous deassertion.

```
    else begin
      R1        <= 1;
      oRstSync  <= R1;
    end
  endmodule
```

The above reset implementation allows a group of flip-flops to be placed into reset independent of a clock but taken out of reset in a manner synchronous with the clock. As a matter of good design practice, the asynchronous assertion–synchronous deassertion method is recommended for system resets.

10.2 MIXING RESET TYPES

10.2.1 Nonresetable Flip-Flops

As a matter of good design practice, flip-flops of different reset types should not be combined into a single always block. The following code illustrates the scenario where a nonresetable flip-flop is fed by a resetable flip-flop:

```
module resetckt (
  output reg oDat,
  input      iReset, iClk,
  input      iDat);
  reg datareg;

always @(posedge iClk)
  if(!iReset)
    datareg  <= 0;
  else begin
    datareg  <= iDat;
    oDat     <= datareg;
  end
endmodule
```

The second flip-flop (oDat) will be synthesized with a flip-flop that has a load or clock enable input driven by the reset of the first. This is illustrated in Figure 10.8.

This requires larger sequential elements as well as extra routing resources. If the two flip-flops are split according to the following code:

```
module resetckt(
  output reg oDat,
  input      iReset, iClk);
  input      iDat);
  reg        datareg;

  always @(posedge iClk)
    if(!iReset)
      datareg <= 0;
```

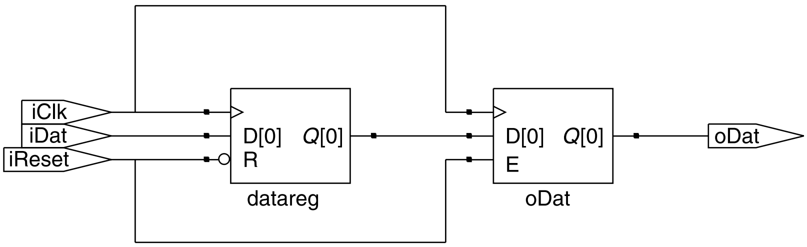

Figure 10.8 Implementation with mixed reset types.

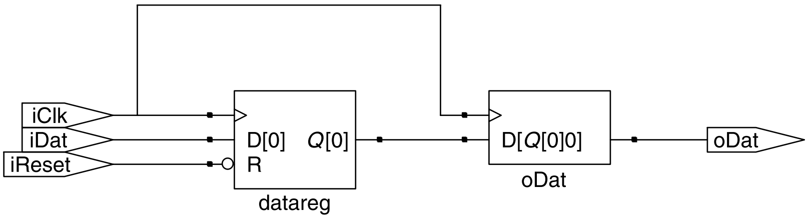

Figure 10.9 Optimal implementation with mixed reset types.

```
        else
          datareg <= iDat;
      always @(posedge iClk)
        oDat      <= datareg;
    endmodule
```

the second flip-flop will have no unnecessary circuitry as represented in Figure 10.9.

Different reset types should not be used in a single always block.

10.2.2 Internally Generated Resets

In some designs, there are conditions where an internal event will cause a reset condition for a portion of the chip. There are two options in this case:

- Using asynchronous resets, design the logic that derives the reset to be free of any static hazards.
- Use synchronous resets.

The main problem with a static hazard on an asynchronous reset is that due to variances in propagation delays, a reset pulse could occur even though the logic is switching from one inactive state to another. Assuming an active low

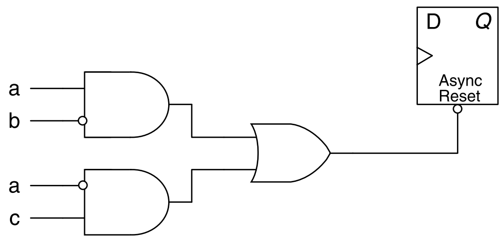

Figure 10.10 Potential hazard on reset pin.

reset, a glitch on an inactive state would be defined as a static-1 hazard. As an example, consider Figure 10.10.

With the circuit in Figure 10.10, a static-1 hazard can occur during the transition (a, b, c) = (1, 0, 1) −> (0, 0,1) as shown in Figure 10.11.

As can be seen from the waveforms in Figure 10.11, a static-1 glitch can occur on the reset line when one of the terms that sets the output inactive (logic-1) becomes invalid before the next term sets the reset high. Remembering back to logic design 101, this can be represented in a K-map format as shown in Figure 10.12.

Each circled region in Figure 10.12 indicates a product term that sets the reset inactive. The static-1 hazard occurs when the state of the inputs changes from one adjacent product term to another. If the first product term becomes inactive before the second is set, a glitch will occur. To fix this problem, a redundant prime implicant is created to bridge the two product terms as shown in Figure 10.13.

By adding the new product term as indicated by the redundant logic mapping, we eliminate the possibility of a hazard that will cause a glitch while the reset is in the inactive state.

In general, the technique for eliminating static-1 hazards will prevent a glitch on an internally generated reset line. However, the use of a fully synchronous

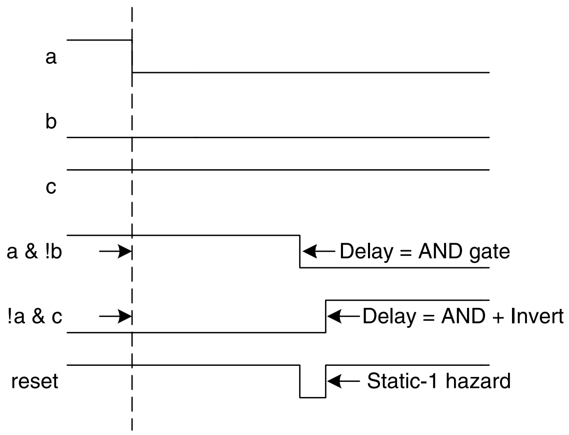

Figure 10.11 Example waveform with reset hazard.

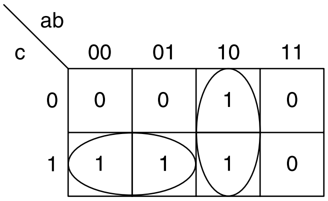

Figure 10.12 Identifying the static-1 hazard.

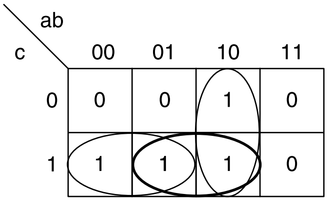

Figure 10.13 Adding a prime implicant.

reset synchronizer is typically recommended in practice. This helps to maintain a fully synchronous design and to eliminate the redundant logic necessary to maintain a glitch-free reset signal.

10.3 MULTIPLE CLOCK DOMAINS

We have already established that reset deassertion must always be synchronous and that asynchronous reset signals must be resynchronized for deassertion. As an

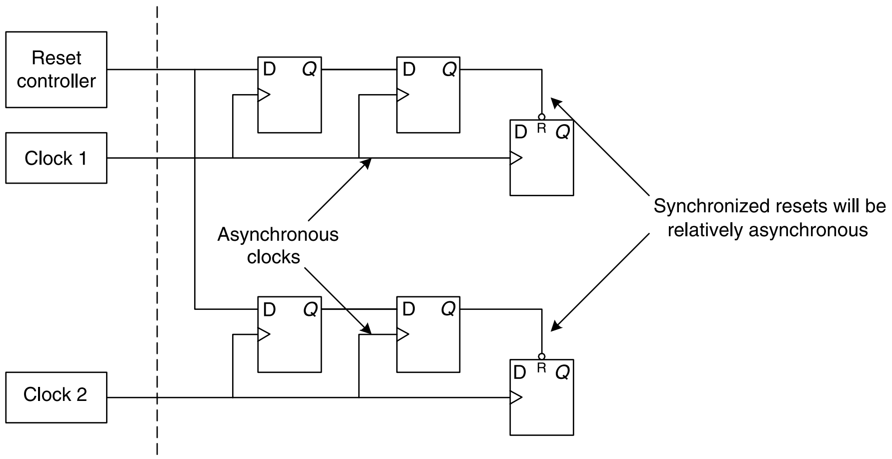

Figure 10.14 Reset synchronization with multiple clock domains.

extension of this principle, it is important to synchronize resets independently for each asynchronous clock domain.

As can be seen in Figure 10.14, a separate reset synchronization circuit is used for each clock domain. This is necessary because of the fact that a synchronous reset deassertion on one clock domain will not solve the clock recovery problems on an asynchronous clock domain. In other words, a synchronized reset signal will still be asynchronous relative to the synchronized reset from an independent clock domain.

A separate reset synchronizer must be used for each independent clock domain.

10.4 SUMMARY OF KEY POINTS

- Reset recovery time violations occur at the deassertion of reset.
- Fully synchronous resets may fail to capture the reset signal itself (failure of assertion) depending on the nature of the clock.
- A reset circuit that asserts asynchronously and deasserts synchronously generally provides a more reliable reset than fully synchronous or fully asynchronous resets.
- Different reset types should not be used in a single always block.
- A separate reset synchronizer must be used for each independent clock domain.