FPGA DESIGN RELIABILITY II

This paper is the second in a series addressing how to build a robust FPGA-based design. Again, what does robust mean? It means designing an FPGA that works predictably and reliably. All the logic, all the clever algorithms, all the computational engines and data communication paths must build upon an FPGA that works reliably due to good design practices.

This issue discusses the concepts of synchronous and asynchronous resets, the proper and improper use of latches,

1. Resetting Synchronous FPGA Elements

1.1. Improper Reset Techniques

In general, the asynchronous reset of flip-flops within an FPGA should be avoided because the timing of the reset signal has no relationship to flip-flop's clock. The designer may not care whether one flip-flop enters a reset state slightly ahead of another. However, the time at which a flip-flop recovers from a reset signal is important.

Consider what happens when a flip-flop's reset is de-asserted at the same instant that the clock edge occurs. Will the flip-flop output take on the value of the data at the input, or will it remain in a reset state? In fact, this situation may produce a metastable event similar to that produced by insufficient data hold time. If the asynchronous reset can be made to de-assert a certain minimum time, known as the "reset recovery" time, before the clock edge, the flip-flop will behave predictably. However, the reset recovery time is unpublished for many FPGA families. As a result, producing a reliable design, while using asynchronous resets, may require some guesswork.

Now consider this problem over multiple flip-flops reset by the same signal. Upon removal of the reset, one flip-flop might clock input data through to the output while another flip-flop might not. If these two flip flops happen to be part of the state encoding for a finite state machine, the state machine may immediately go to an invalid state, resulting in a severe error.

1.2. Design Guidance: Use of Synchronous Resets

Wherever possible, a designer should use synchronous resets. As the name implies, the synchronous reset input to a flip-flop is only sampled on the clock transition. Like a data input, the synchronous reset has setup and hold time requirements, although they may have different values than those of the data input. In some FPGA families, the flip-flops do not have dedicated synchronous reset inputs. In this case, when a designer specifies a synchronous reset, the combinatorial array or look-up table associated with the flip-flop is used to construct logic by which the flip-flop is forced low when the reset is asserted. In this case, the synchronous input

becomes just another logic term that comprises the data input. The implementation software calculates timing for the reset, as it would for any other data input.

1.3. Design Guidance: Proper Use of Asynchronous Resets

Many FPGA flip-flops have dedicated asynchronous reset inputs. These may be tied together and used globally (e.g. to hold all flip-flops in an initial state during FPGA configuration), or used locally. We will examine both situations.

Use of an externally supplied global asynchronous reset may be beneficial if the FPGA clock is absent at system startup. However, this has the disadvantage of allowing different synchronous elements within the FPGA to recover from reset – when the reset is de-asserted – at different times. A solution that overcomes the problems of each approach is to assert the global reset signal asynchronously and to de-assert it synchronously with the clock. The internal global reset must be the logical OR of two signals. The first is the external reset; the second is an internally generated reset that de-asserts only after the clock is running and the external reset is removed. The latter component must be generated from flip-flops, so that its de-assertion is synchronous with the clock.

A global asynchronous reset is often used in an asynchronous reset clause at the beginning of a VHDL process. This establishes the power-up and reset state of the FPGA flip-flops synthesized from that process. Either the global reset should be de-asserted synchronously as previously described, or each process should initialize to an idle state, in which no activity takes place. Exiting the idle state should occur only due to an event that takes place after reset has been de-asserted.

When local asynchronous resets are designed into a logic circuit, FPGA implementation software typically does not provide any guarantee that metastability will be avoided. For this reason a designer should design with synchronous resets whenever possible. Nevertheless, some logic functions cannot be constructed without the use of an asynchronous reset. An example was previously discussed (see “FPGA Design Reliability II”) The example illustrates one of the important principles when using a local asynchronous reset:

- If a flip-flop is subject to metastability for any reason (including use of an asynchronous reset), the metastability must be isolated from other circuitry by using a synchronizer.

Other principles for asynchronous reset use are:

- Synchronizing an asynchronous reset pulse (by passing it through a pair of clocked flip flops) is useful to prevent a glitch from occurring upon de-assertion of the reset (the trailing edge of the pulse). Keep in mind that it’s the trailing edge of the reset that’s important.
- The asynchronous reset pulse must be wide enough to ensure the flip-flop is actually cleared. The minimum width is published by the FPGA manufacturer. Failure to meet the requirement may result in metastability of the flip-flop.

- Flip-flop recovery time from an asynchronous reset is often unpublished. However, it may be possible to assume the recovery time to be the same as the time required for the reset, when asserted, to take effect (e.g. reset to output low). This assumption should be verified with the FPGA manufacturer whenever possible.

2. Latch Use

When using HDLs (e.g. VHDL or Verilog) for design entry, it is possible to write seemingly innocent code which, when synthesized, creates a latch in the FPGA. Often, such latch synthesis is inadvertent; the designer did not intend there to be a latch. The latch may be built out of combinatorial logic, or a flip-flop may be configured to operate as a latch (a feature of some Xilinx families). In either case, the resulting latch is subject to race conditions not present in a flip-flop. These race conditions may produce particularly unexpected results when the data and gate signals of the latch are generated by upstream synchronous logic. In most cases, latch use in an FPGA should be avoided. In addition, a designer must learn to recognize HDL code that will infer a latch when synthesized.

2.1. Design Guidance: Proper Use of Latches

There is little, if any, benefit to using a latch within the core of an FPGA. Latches should only be considered when processing external inputs to the FPGA. As an example, designers familiar with the timing of the address lines of the old ISA bus realize that latches are necessary to achieve the highest performance interface. Most FPGA manufacturers provide the ability to configure I/O flip-flops (flip-flops within the I/O structures that ring the device die) as latches. Where latches on inputs are required, these are the resource of choice. Most manufacturers discourage building latches from FPGA combinatorial logic, since it is slower and more prone to race conditions and oscillation than an I/O flip-flop configured as a latch.

2.2. Design Guidance: Avoiding Latch Inference

A number of HDL coding errors can result in the unintended inference of a latch. We will examine the more common errors in this section, using VHDL examples.

2.2.1. Latch Inference through Improper Use of Variables

The intended purpose of a VHDL variable is as an intermediate value within combinatorial logic. Within a VHDL process, variables are often used incorrectly, causing an unexpected latch to be synthesized. In Figure 2.2-1, the variable *incorrect_v* is read before it is assigned to. The VHDL code for the clocked process is shown at the left of the figure. *incorrect_v* uses its previous value, thus inferring a register. Had this been a combinatorial process, a latch would have been inferred in place of the flip-flop.

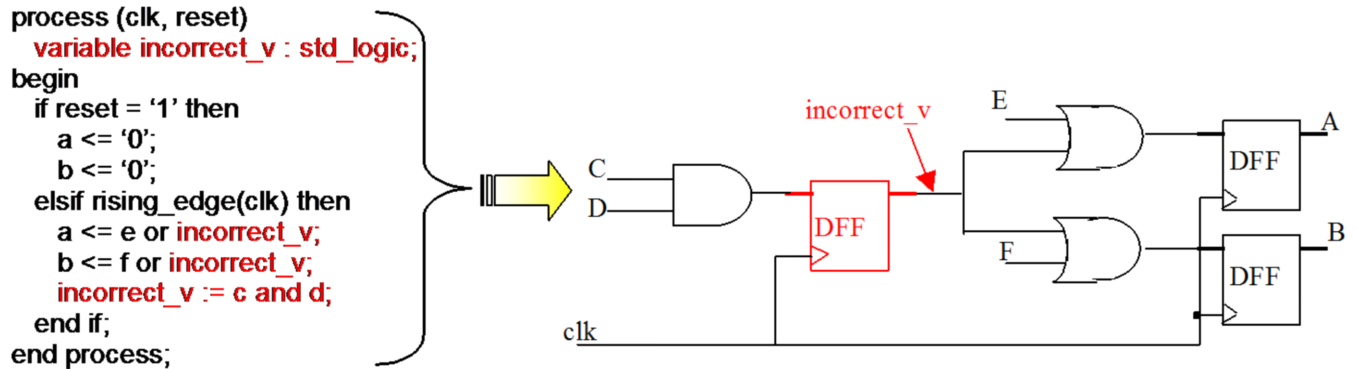

Figure 2.2-1 Incorrect Use of Variables

Always make an assignment to a variable before it is read. Otherwise, variables will infer either latches (in combinatorial processes), or registers (in clocked processes) to maintain their previous value.

2.2.2. Latch Inference through Inadequate Branch Coverage or Inadequate Signal Assignment

In this situation, latches are inferred for two primary reasons: not covering all possible branches in if-then-else statements, or not assigning to each signal in each branch. This is only a problem in a *combinatorial process.*

A latch inference example for each of these cases is shown in Figure 2.2-2. The schematic for code example (1), generated during an actual synthesis of the code, is shown. The schematic for code example (2) is not shown, but would be similar. The blocks named *latrs* in the schematic represent the inferred latches.

For code example (1), every possible state was not covered. This is a very common mistake for one-hot encoded state machines – those states that are not "one-hot" have no coverage by the "if" clauses (see section 2.2.3.1.1 for a discussion of one-hot finite state machine encoding). In this case, latch inference would be eliminated by the use of a final '*else'* statement containing an assignment to the signal *cs.*

For code example (2), the latches are inferred because not every signal is assigned in each state.

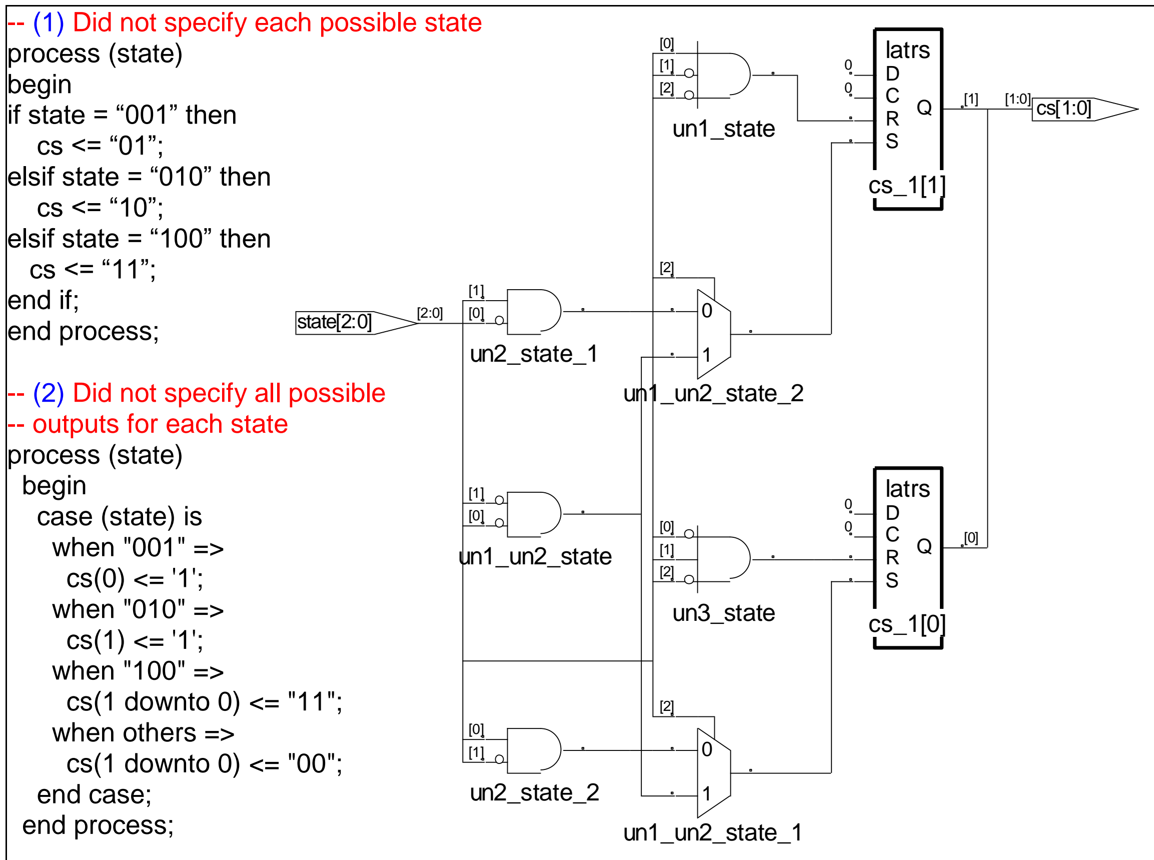

Figure 2.2-2 Latch Inference

Figure 2.2-3 shows the correct code, which eliminates latch inference by covering all possible branches and by assigning to every signal in each branch. The corrections are highlighted in blue. For the if-then-else statement, adding the else clause solves the problem. For the case implementation, the default assignment to *cs* before the case statement specifies a default assignment for each state. When using this code construct, only signal assignments that deviate from the default state are required. Using a default assignment is equivalent to making exhaustive signal assignments in each *when* clause.

```
(1) Fixed if-then-else implementation
process (state)
begin
   if state = "001" then
      cs <= "01";
   elsif state = "010" then
      cs <= "10";
   elsif state = "100" then
       cs <= "11";
   else
       cs <= "00";
   end if;
end process;
```

```
-- (2) Fixed Case implementation
process (state)
begin
    cs <= "00";
    case (state) is
        when "001" =>
           cs(0) <= '1';
        when "010" =>
           cs(1) <= '1';
        when "100" =>
           cs <= "11";
        when others =>
           cs <= "00";
end case;
end process;
```

Figure 2.2-3 Elimination of Inadvertent Latch Inference

2.2.3. Additional Design Guidance

2.2.3.1. Finite State Machine Design

Producing reliable finite state machines (FSMs) requires planning, attention to detail during the design process, and thorough simulation. This section discusses a few FSM considerations of special interest.

2.2.3.1.1. FSM Encoding Style

The states of FSMs must be represented by state variables in order to be synthesized into logic. The FSM states can be encoded by the state variables in a number of ways, including:

- Binary or maximal encoding: uses the fewest bits required to represent all of the FSM's states.
- One hot encoding: dedicates a separate flip-flop to each state. A flip-flop is set, while all others are cleared, to uniquely identify a state.
- Gray encoding: a binary representation with the goal of having only one bit transition between adjacent states.
- Zero hot encoding: similar to one hot encoding, but with a cleared flip-flop uniquely identifying each state.
- Safe encoding: Provides recovery from any undefined or illegal state. This is typically used with one hot encoding.

FPGAs have a large number of registers, but rather limited combinatorial logic in front of each flip-flop. Large combinatorial functions, built by interconnecting multiple look up tables (LUTs), are relatively slow. Therefore, for high-speed performance in FPGAs, FSMs should be implemented in a way that will minimize combinatorial logic, with a relative disregard for the

number of flip-flops consumed. Combinatorial logic is reduced when flip-flop fan-in (the number of signals combined into the data input of a flip-flop) is reduced. Fan-in reduction is the goal of one hot and similar styles of encoding. One hot encoding works well to reduce fan-in if the number of paths into any state in the FSM is very small. An FSM having a state transition diagram (bubble diagram) that is a unidirectional ring lends itself well to one hot encoding. Because most states in many FSMs have few input paths, one hot encoding (or a similar style) is usually the style of choice in FPGAs. FSMs with more complex topologies may benefit from other encoding styles

CPLDs, on the other hand, have relatively few flip-flops, but allow quite complex signal fan-in to each flip-flop with less of a performance penalty than in an FPGA. In this situation, a form of binary encoding often produces the best result.

In particularly demanding situations, a designer may need to handcraft the encoding of state variables in a way that does not conform strictly to any of the styles listed above, in order to achieve the desired timing performance. This may result in a mixture of styles combined in a single FSM. One state may be assigned a dedicated flip-flop, while other states are grouped together and binary encoded. It's also possible in some cases to reduce fan-in by using registered output variables in the combinatorial logic that decodes the FSM's next state.

All the techniques discussed so far are possible if the designer explicitly specifies the encoding of each FSM state. This can be done with HDL and schematic entry. However, HDL entry allows *symbolic* representation of the states (e.g. using enumerated types in VHDL), in which no specific encoding is dictated. The designer then makes his or her wishes known to the synthesis tool, by in-line synthesis directives, or by means of switches in either the command line or GUI. With some software, the synthesis software has the ability to choose the encoding style for each FSM. Modern synthesis tools have become quite sophisticated; they are able to analyze the FSM code, and, in most cases, choose the optimal style. If a designer chooses this option, he or she can code all FSMs using timesaving symbolic encoding, then allow the synthesis tool to choose the optimal encoding style for each one. While this works well for the vast majority of cases, synthesis tools cannot yet mix and match encoding styles and use the other "tricks" required of the most demanding cases, previously discussed.

2.2.3.1.2. FSM VHDL Processes

It is common to code FSMs using one, two, or three VHDL (or Verilog) processes. Most synthesis tools suggest coding state machines with three process statements, one for the next state decoding, one for the output decoding, and one for registering of outputs and state bits. This isn't as concise as using one process statement to implement the entire state machine, however it allows the synthesis tools the ability to better optimize the logic for both the outputs and the next state decoding. Another popular style is to use two processes to implement the state machine, one for next state and output decoding, and the other process for registering of output variables and state variables. Both two-process and three-process FSMs generally produce more readable and maintainable code than FSMs coded with a single process.

Moore state machines (output is dependent only on the current state), generally have limited decoding for the outputs and the state machine can often be cleanly coded with only one process

statement. Mealy state machine (outputs depend on the inputs and the current state) output decoding is generally more complex, and therefore the designer should use two or three processes.

2.2.3.1.3. FSM Latch Inference

Designers should take care to avoid latch inference in an FSM. These errors are most likely to occur when FSMs are coded with two or three VHDL processes, because of the separate combinatorial process in which latch inference is possible. This can be avoided by following the guidelines given in section 2.2.