Understanding Recovery and Removal Analysis

03/25/2008 version 1.1

TimeQuest automatically does recovery and removal analysis, which is a type of static timing analysis that many users are unfamiliar with. The purpose of this document is to briefly describe what is being analyzed, as well as what a failure might look like in a user design, and some suggestions on what a user can do to meet timing.

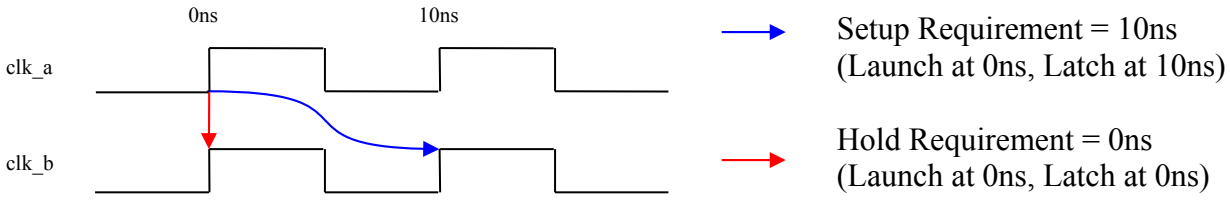
From a high-level, recovery and removal ensure that the logic comes out of the reset state together, so on one clock edge every register is held in reset, and by the subsequent clock edge, every register has been released from reset.

What is Recovery and Removal analysis?

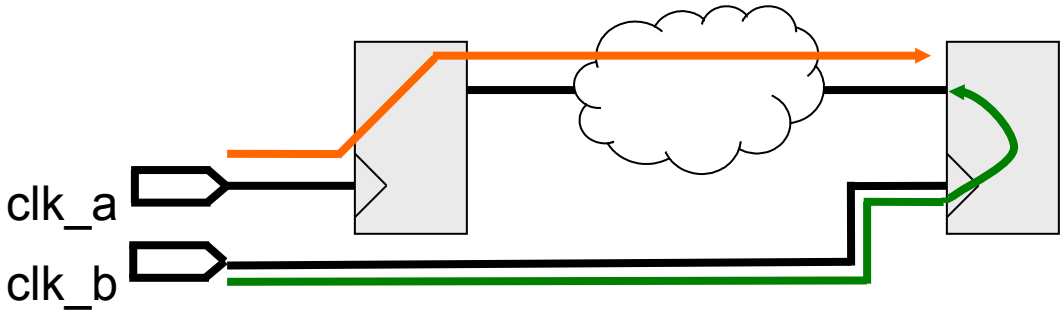
The best way to explain recovery and removal analysis is to compare it to something users generally do know, setup and hold. So starting with user constraints in their SDC file:

create_clock -period 10.000 -name clk_a clk_a
create_clock -period 10.000 -name clk_b clk_b

We have two clocks with the same period, 10ns. This results is a setup requirement of 10ns, and a hold requirement of 0ns.

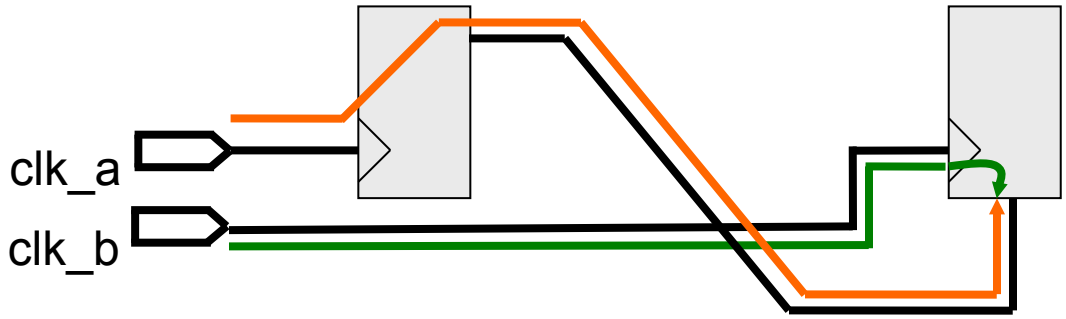


The paths in a user design are then analyzed like below, where the orange line drives a synchronous port on the destination register, such as the data input, clock enable, or synchronous clear.



For most paths in a user design, clk_a and clk_b are the same clock and on a global. So the clock delays to the source and destination registers come close to cancelling each other out, and the basic requirement is that the data path between the two registers must be shorter than 10ns to pass the setup requirement, and greater than 0ns to pass the hold requirement. Make sure you understand this, as it is a very fundamental principle of timing analysis.

Based on that premise, the ONLY difference for recovery and removal is that the orange line feeds an asynchronous port on the destination register rather than a synchronous port, and timing analysis is done when the reset de-asserts itself, letting the destination register come out of reset. Everything else is identical.



Once again, in the simpler common case, if clk_a and clk_b are the same clock and on a global, i.e. there is little clock skew, then the requirement is that the path from the source register to the destination register's asynchronous input is less than 10ns for recovery analysis, and greater than 0ns for removal analysis.

I did not draw the "cloud of logic" on this path either, just because it is generally recommended against having any logic on the reset path. So what we've found is that:

Recovery Analysis is analogous to Setup Analysis, except the data path feeds an asynchronous port on the destination register.

Removal Analysis is analogous to Hold Analysis, except the data path feeds an asynchronous port on the destination register.

As can be seen with static timing analysis, it is easiest to think of the asynchronous clear and preset as if they were synchronous signals and just labeled recovery and removal instead of setup and hold. Later on we will cover why we don't just make them synchronous.

What do the names Recovery and Removal mean?

When the asynchronous signal is de-asserted, the removal ensures that it is late enough after the required clock edge, keeping the flip-flops removed from normal operation, and the recovery analysis ensures that it de-asserts before the required clock edge, allowing the registers to recover into normal operation.

Looking back at the waveform on the first page, when a reset de-assertion is launched at time 0ns, recovery and removal ensure that this signal reaches all of its destination registers between the red and blue lines, and all registers come out of reset on the same clock cycle.

What does a Recovery failure look like?

I find it useful to describe what a failure looks like in hardware for users to fully appreciate recovery and removal. Let's start with recovery, as that is by far the most common type of failure, just like setup failures are the most common inside an FPGA.

Let's look again at the reset from a design perspsective:

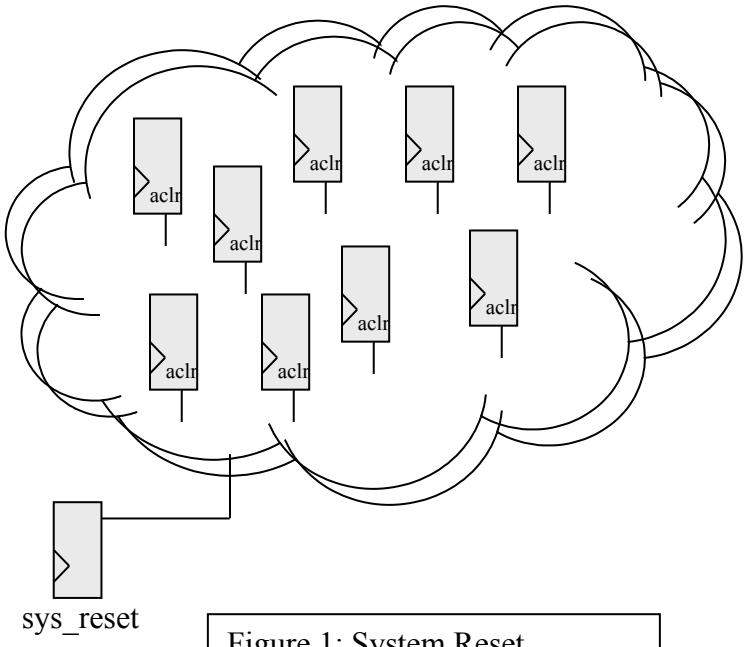


Figure 1: System Reset

The sys_reset generally feeds all the registers in a clock domain(or group of related clock domains), which may vary from a handful of registers, to over a hundred thousand registers. If sys_reset is asserted, recovery analysis is when the sys_reset de-asserts itself and all of the destination registers come out of reset into active mode. Now, if the delay from sys_reset to all the destination registers occurs within one clock cycle, then all of the registers come out together and they all see next clock edge together. But if some registers fail recovery time, they might still be in reset on that following clock cycle and therefore not see that clock edge.

So how does this actually cause a failure? Let's say some of these registers were in a state-machine that resets to a state called *powerup*, but on the first clock moves to a state called *idle*. Now, if some of the state-machine registers see that first clock cycle and some do not, then some registers switch to the idle state while other registers do not. This could cause the state-machine to go to the wrong state, and possibly an unknown state that it might never recover from. This would be a recovery failure, caused by the asynchronous reset signal not de-asserting its destination registers in time.

Another scenario is a design with counters that immediately start counting out of reset and are supposed to be in-sync with each other. If one of the counters misses the first clock out of reset due to a Recovery failure, then they will not be in sync and the design no longer works as expected.

Are most structures susceptible to Recovery failures?

There are two basic requirements for a recovery failure to affect the design:

1) The logic must "change state" on the first clock cycle/s
2) That "new state" must be important

The first criteria is relatively straightforward. If the user's logic does not change value on the first clock cycle/s, then it doesn't matter whether it sees those clocks or not. If a state-machine sits in its reset state out of power-up, then a recovery failure to some of its registers wouldn't matter, since those registers wouldn't change anyway. Most registers, especially control registers, do not change state on their first clock cycle.

The second criterion is a little more complex. Usually anything in the data path is ignored on the first few clocks. For example, an adder may have a recovery failure and add to the wrong value, but if the system is ignoring the data out of power-up and waiting for good data to come through, then that bad addition does not matter and the recovery failure will not affect the system.

On most designs, almost all of the logic is immune to recovery failures, as most logic will work fine even if it fails recovery timing. Some designers consciously make sure they don't put in any logic where a recovery failure would cause problems, which is generally quite easy to do. That being said, if the small portion of a design that is susceptible to recovery failures experiences one, then the design will fail.

These failures are especially aggravating because they are inconsistent. If a design fails recovery by 500ps, most devices will not come out of reset under the worst case conditions of Process, Voltage and Temperature and work fine. But a handful of devices will experience occasional reset failures in the field. There is no way to simulate recovery failures because there is no way to say exactly which registers miss the first clock cycle and which ones do not. I have seen many users spend a LOT of time debugging these issues(when they did not do any recovery/removal analysis to begin with), and it would have been much easier to close timing on recovery and removal to begin with.

What does a Removal Failure look like?

To be honest, I have never seen a removal failure. Removal analysis should always meet timing, as long as the user is not delaying the clock to the destination register, which is usually done with a gated clock or a clock shift.

An example of what a recovery failure might look like is if the destination clock is gated, and therefore it received its clock edge later than the sys_reset source register does. When the sys_reset de-asserts, it may release some registers before the same clock edge that triggered sys_reset, much like what a hold violation looks like. The net result is similar to a recovery failure, in that some registers come out of reset a clock earlier than other registers. Similarly bad things can happen to logic that transitions on that first edge.

Why are we timing an asynchronous circuit?

This is actually the first question most users ask, but having explained how a failure occurs, it hopefully makes sense why these requirements exist. Now that we know what a failure looks like, and that essentially we are timing it as if it were synchronous, probably a better question is:

Why not just make the reset synchronous?

There are two reasons for this. The first is to more effectively utilize the device resources. The asynchronous reset/preset already exists on each FPGA register, whether it is used or not. If the designer made their reset synchronous, then they have to use one of the data inputs into the FPGA, which can hurt timing and area. For example, if a Cyclone III design, which has 4-input LUTs, had a 4-input AND gate feeding a register, then changing the register's reset from being asynchronous to synchronous would force another input into this function. The logic would require 5-inputs, forcing it to use another LUT, making that data path both larger and slower. If the clock enable were used on this register also, that enable signal would also have to be gated with the synchronous reset(since the reset has priority), possibly making that path larger and slower too. So from a resource perspective, it makes sense to use the "free" asynchronous ports in the FPGA.

The other reason resets are designed asynchronously is to make them more robust. By being asynchronous, they can reset the design without requiring a clock. Although some designs do not have this requirement, many do, and being able to reset the design when the clock is disabled may be an important consideration. Let's look at the following reset structure as an example of a design that can reset without a clock, but comes out of reset with a clock:

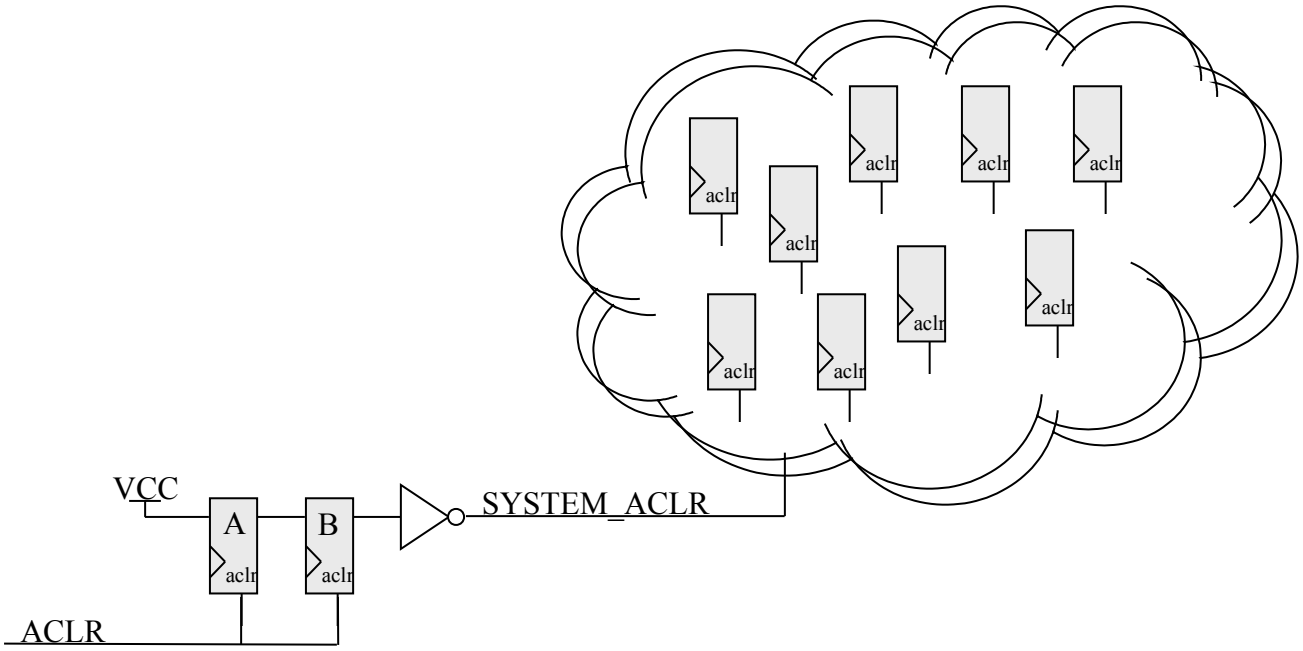


Figure 2: Example Reset Structure

In the above diagram, the ACLR is generally asynchronous, which is why we have two(or more) registers for metastability. When the ACLR goes active to remove the design from operation, it goes through the B register in an asynchronous manner, and resets all of the destination registers, whether or not there is a clock. But when the ACLR is released, the system logic recovers after two clock cycles, allowing the VCC to pass through registers A and B in a synchronous manner. For constraining this circuit:

- The ACLR net should not be timed if it is truly asynchronous. A set_false_path assignment may be necessary.

- The path between registers A and B can get a set_max_delay assignment that is less than the clock period, ensuring the path meets timing with margin and reducing mean time between failures(MTBF). MTBF is a metastability topic which won't be discussed here.

Note that although this is a common reset structure, it is by no means the only or best asynchronous reset structure. Many users have their own methods, and this example is purely shown on how a system can be reset without a clock and comes out of reset with a clock.

What if I don't make Recovery timing?

If a system does not meet recovery timing, there are multiple things a user can do to close timing. The most important thing is to first understand why the design is failing timing. Let's look at a "common" scenario for meeting timing, before looking at failures.

A common system has a reset register that fans out to the asynchronous clear/preset port of many registers(See Register B in Figure 2 above, for an example). This register usually drives a global to easily fan-out across the device. This is a very common scenario, but failures can occur if:

a) The source register is not placed near the global driver. With timing requirements, the fitter should place the register near the location it gets onto the global, but if this doesn't occur, there will be a long delay to all of the registers. It is always worth checking that the delay to the global is not long. (Globals are somewhat slow paths because they are large, low-skew drivers that fanout across the device, so be sure not to confuse the path to the global with the actual global itself.)

b) The net is not placed on a global and takes too long to cross the device. For example, if the clock domain is 4ns, it may take more than 4ns to get across a large device, and therefore the path fails recovery.

c) The net is on a global, but the global delay is actually too slow compared to the clock rate. For example, in an EP2S180-5, it may take more than 4ns to get across the chip on a global, so the raw delay would fail timing.

d) The register is crossing clock domains, and so the recovery requirement is not realistic. For example, if the reset register is driven by an 8ns clock, and the destination register is driven by a 10ns clock, the requirement will end up being 2ns.

So what can a user do? Here are some suggestions, which must be balanced against what is going wrong:

a) If driving a global and the reset register is not placed near the global driver, then the user should add a location assignment to that register, forcing it near the global.

b) If the asynchronous reset fails timing and does not use global routing, try assigning it to use a global(In the Assignment Editor, make a Global = On assignment to the source register driving the asynchronous signal, e.g. register B in Figure 2)

c) If the asynchronous reset fails timing and uses global routing, try the using local routing. (In the Assignment Editor, make a Global = Off assignment to the source register driving the asynchronous signal, e.g. register B in Figure 2)

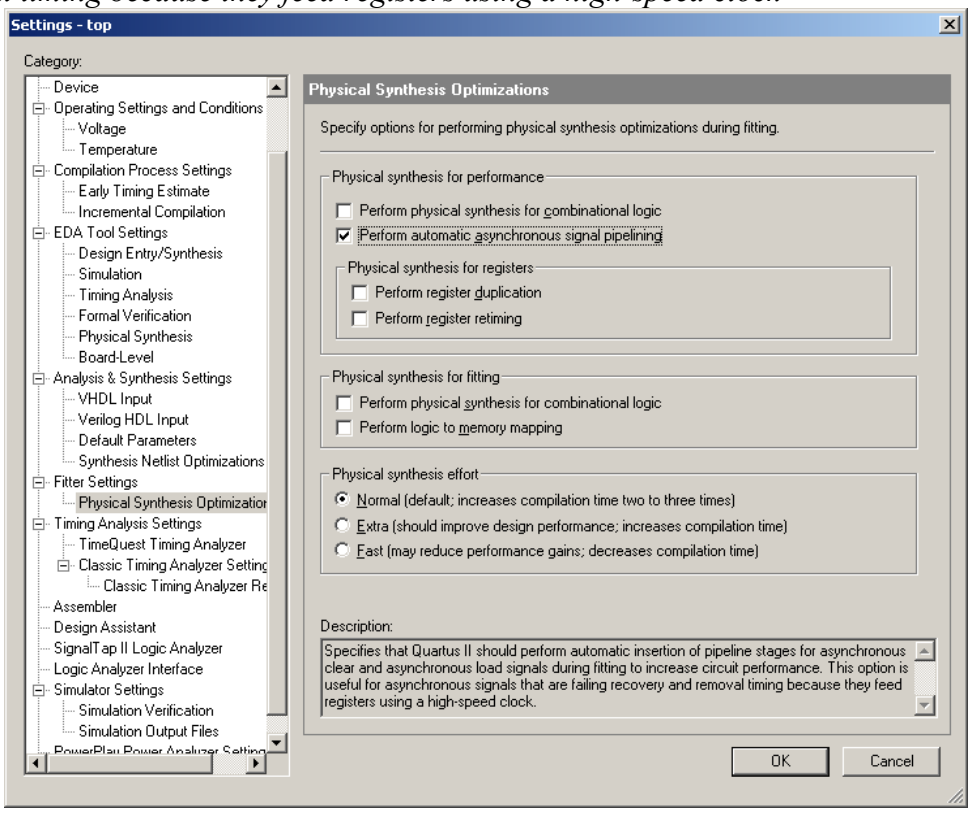
Note that b) and c) are exact opposites of each other. If the asynchronous clear does not drive a lot of registers, it is often faster to use local routing, since it can be placed near the registers it drives and get to them very quickly. If the asynchronous clear drives a lot of registers that are spread across the device, then using a global is often faster. There is no exact science for this, which is why both are being recommended.

d) If crossing clock domains, re-register in the new clock domain so that the recovery requirements are not unreasonable. In general, a separate reset structure should be used for each group of synchronous clock domains. (If a PLL creates related clocks with periods of 20ns, 10ns and 5ns, then just a single reset structure could be used for all three domains, since their edges are aligned. But if it creates clocks with periods of 10ns and 6.666ns, then a separate reset structure should be used for each domain)

e) If using local routing, duplicate the reset structure in large sub-hierarchies. A good example of this is shown in *Application Note 470: Best Practices for Incremental Compilation Partitions and Floorplan Assignments*. Search on "Cascaded Reset" to show the example. With a cascaded reset structure, the fan-outs of each reset will not span the chip, and it is recommended to not use globals for these resets, as they will quickly use up this valuable resource. This is often the only solution for high-speed domains. Note that almost all designs don't care if different hierarchies come out of reset on different clock cycles, as long as all the registers within that hierarchy come out at the same time. So if a design uses one register to feed a top-level hierarchy, but has two cascaded registers to feed another hierarchy, it should be fine. If there is a problem, since the design meets timing, it will show up in simulation too.

f) If the logic is recovery-immune, add set_false_path assignments. For example, if a user's state-machine is failing recovery timing, but the designer is confident it is immune to recovery failures, adding a *set_false_path -from reset_register -to *state-machine** will cut it from timing analysis. It can be quite difficult to find all the logic that is immune to recovery errors, so this is generally not a recommended solution. If a customer is confident that their entire design is recovery/removal immune, then they can cut the entire net with a *set_false_path -from reset_register*.

g) Turn on the Physical Synthesis option Perform Automatic Asynchronous Signal Pipelining(see screenshot below). This is under Assignments -> Settings -> Physical Synthesis, and the description is given as:
Specifies that Quartus II should perform automatic insertion of pipeline stages for asynchronous clear and asynchronous load signals during fitting to increase circuit performance. This option is useful for asynchronous signals that are failing recovery and removal timing because they feed registers using a high-speed clock



What if I want to use my asynchronous ports for logic instead of a system reset?

Some designers do this, most often in schematics, since the asynchronous port is visibly sitting there. For example, if some condition occurs and the user wants that to reset a counter, they may hook it up to the aclr port of the counter. This is strongly recommended against, as the asynchronous ports in a design are not intended for this type of logic, and instead the designer should use a synchronous port. Technically, in this example, the user would need to time the reset assertion not only to the counter, but through the counter to the destination registers it feeds. When the signal de-asserts, they would then only time it to the counter. This is not how static timing analysis tools work and the path through the register will not be analyzed, as that is not the intent of asynchronous ports in the FPGA.

Conclusion

Although this may seem like a lot of information, most designs meet recovery and removal timing without any user intervention. It is important that the designer think about their reset structure early on, and design it for their system requirements. This is usually not too difficult, and as an example, many designs only require that the reset be synchronized (double-registered) when feeding unrelated different clock domains. Once that is done, everything falls into place and meets timing on its own.