FishTail: The Formal Generation, Verification and Management of Golden Timing Constraints

Chip design is not getting any easier. With increased gate counts, higher clock speeds, smaller chip sizes and reduced power requirements, designers have a very difficult task. Today's virtual prototyping and chip-implementation tools are powerful and address several key deep-sub micron issues, but there remains a fundamental conundrum. Precise constraints on chip timing, upon which the design ultimately succeeds or fails, remain in a state of flux throughout the design cycle. False paths and multi-cycle paths are typically entered only in response to timing problems. As timing problems seriously manifest themselves only during place & route, this is late in the design cycle to be tweaking your fundamental timing goals. All of this results in extra timing closure iterations and longer turn-around-time, the risk of silicon failure because of incorrect timing constraints entered by design engineers, and a messy handoff from chip design to implementation teams.

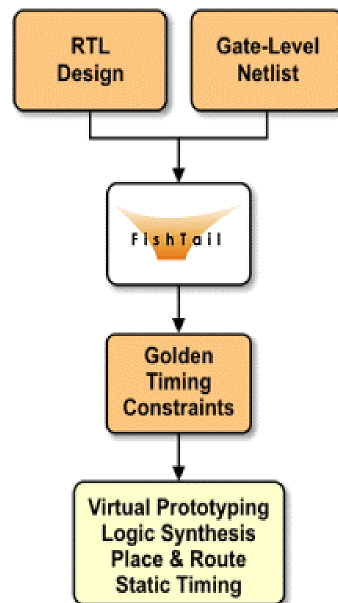


Figure 1: The Formal Generation, Verification and Management of Timing Constraints.

FishTail Design Automation has developed ground-breaking, patented technology to solve this problem, reducing risk and improving design quality.

Products

FishTail's design constraint generation product, **Focus**, starts with the RTL or netlist description for a design. Focus generates a template design constraint file that points out the nets on the design on which clocks and generated clocks should be defined and creates default input/output delays for the ports on a chip. This template constraint file is then massaged by users to generate the final clock and boundary constraints for the chip. Next, the design constraints and RTL/netlist are input to Focus and used to synthesize timing exceptions (false and multi-cycle path definitions). False paths synthesized by Focus result from asynchronous clock-domain crossings, the multiplexing of clocks in the clock generation circuitry and the

manner in which combinatorial control logic affects the flow of information on a chip. Multi-cycle paths synthesized by Focus result from the use of control logic to cause additional clock cycles to be inserted when propagating information from one register to another. Users can automatically have the tool generate separate constraint files for the different place-and-route blocks on a chip from a single chip-level run.

FishTail's timing-exception verification product, **Confirm**, reads in the RTL and user-specified timing exceptions for a design. Confirm formally establishes if all the paths constrained by a timing exception are indeed false or multi-cycle. If mistakes are found then Confirm is able to re-write a timing exception more precisely so that all of the paths constrained by the exception are indeed false or multi-cycle. Confirm also supports an assertion-based verification flow for timing exception verification. With this flow, Confirm is used to generate an assertion that states the property that would need to be satisfied for a timing exception to be correct. The assertions generated by Confirm are imported into functional simulation tools and verified by running the RTL regressions for the design.

FishTail's constraint management product, **Refocus**, is used to map the golden timing constraints for a design to gate-level netlists generated by each step of the implementation flow. The intent is to maintain a single repository of design constraints and to use Refocus to map these golden constraints to the netlist as it evolves through the chip-implementation flow. As part of its constraint management capabilities Refocus is able to promote block design constraints to the chip-level. This allows constraints used to drive block-implementation to be reused during full-chip static timing signoff. Refocus handles hierarchy changes, names changes during chip implementation and, if required, can read in register mapping information established by formal equivalence checking tools.

Value Proposition

Customers derive substantial value by deploying FishTail products in their design flow. FishTail products allows them to:

1) Eliminate the risk of silicon failure that results from the application of incorrect timing exceptions. When engineers are under pressure to tape-out a chip they make mistakes and enter timing exceptions that are either incorrect, or broader in their scope than is legitimate. The automated verification of user timing exceptions using Confirm is a formal approach to timing closure that ensures that timing constraints go through similar levels of scrutiny that other aspects of chip design and implementation already do. Our experience has shown that when engineers enter timing exceptions manually they almost always make mistakes. These mistakes are sometimes caught late in the design cycle - during gate-level functional simulation - necessitating time-consuming ECOs. Often, incorrect timing exceptions only manifest themselves in the form of silicon failure of a prototype chip in the lab or of a production chip in the field.

2) Improve the quality of results (QoR) of the final chip implementation. Timing relaxations focus the attention of place and route tools on the real timing challenges on the design, and stop their distraction with the optimization of paths that are false or multi-cycle. As a result, the overall timing of the chip is significantly improved, with modest improvements in the area and power consumption of the chip. Our experiences

have shown total negative slack on designs drop by 25-75% when Focus generated exceptions are introduced into the implementation flow. On designs that already meet timing with user constraints we have seen area reduction of between 3-10% from the introduction of Focus generated exceptions. It is important to appreciate that most of the QoR improvements from applying Focus generated exceptions come not merely from making critical timing paths false, but by directing the attention of implementation tools on the real timing challenges on a design – timing exceptions that apply to both critical and non-critical paths are useful and important from this standpoint.

3) Reduce the time spent in the back-end design flow to close timing. The identification of timing exceptions early in the design flow reduces the number of timing problems that need to be manually addressed during timing closure. This, in turn, reduces the time taken to close timing because there is less back-and-forth between implementation and design engineers in establishing whether timing problems are real or not. The time taken to resolve back-end timing problems is particularly significant when chip-design and chip-implementation teams span geographical and business boundaries. Driving chip-implementation with a complete set of timing exceptions can easily reduce turn-around-time by 4-8 weeks.

Timing Exception Generation Flow

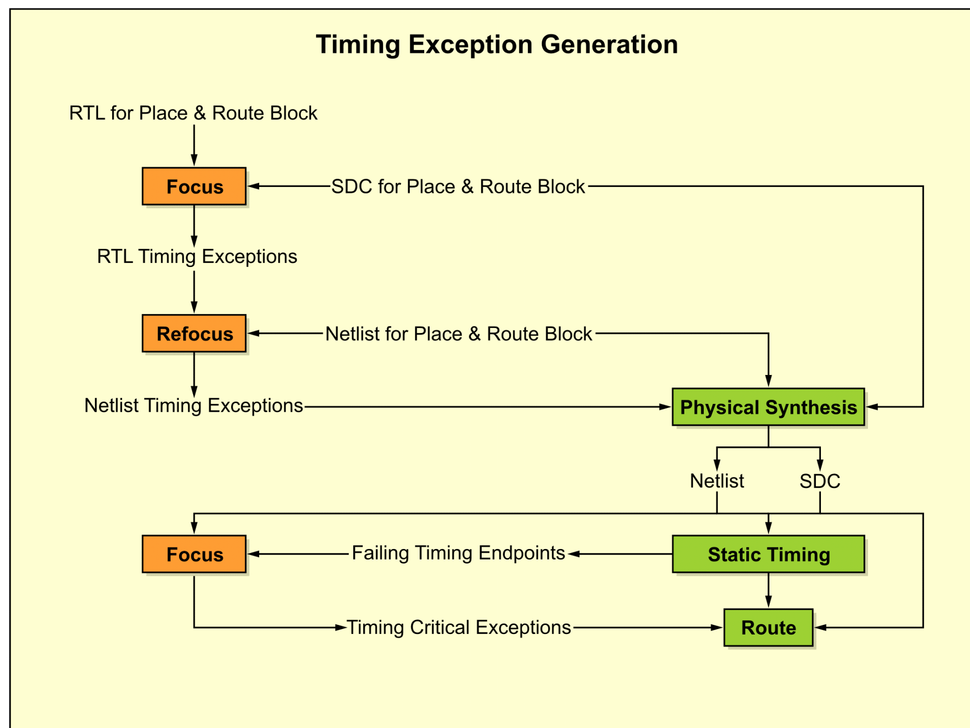


Figure 2: Timing Exception Generation Flow

The Timing Exception Generation Flow, shown in Figure 2, commences by using Focus to generate false and multi-cycle paths for a design. We recommend providing Focus with synthesizable RTL (Verilog, VHDL or a mix) for a physical synthesis block, .lib models for memories, and simulation models for standard cells instantiated in the RTL. If some of the

blocks in a design only have gate-level netlist descriptions then the netlist can be provided as input to Focus. In addition to the design description, Focus reads in the existing SDC constraints for the physical-synthesis block. These constraints specify clocks, I/O delays, case analysis and any existing user-specified timing exceptions. Using this information Focus formally discovers false and multi-cycle paths on a design. All timing exceptions generated by Focus refer to clocks, registers and hierarchical pins on a design.

All exceptions generated by Focus are formally verified by Confirm as part of the exception-generation run. The exceptions generated by Focus are mapped to the netlist provided as input to physical synthesis using Refocus. Refocus automatically handles hierarchy removal and name changes between RTL objects and netlists. If necessary, users can provide Refocus with the mapping information established between RTL and netlist using formal equivalency checkers.

Physical synthesis proceeds using the Focus generated timing exceptions plus the existing user design constraints. At the end of physical synthesis there may be some timing endpoints that do not meet timing. A list of these timing-critical endpoints is generated from within static-timing tools using FishTail provided utilities. Focus is now run at the gate-level using the post-physical-synthesis netlist and SDC plus the list of timing-critical endpoints. The objective of the gate-level Focus run is to generate false paths that apply to timing-critical portions of a design. The false paths that Focus generates when analyzing a gate-level netlist refer to clocks, registers, hierarchical and standard cell pins. The ability to refer to standard-cell pins is something Focus could not do when running at the RTL and so the gate-level false paths are additive to the RTL false paths.

The timing-critical false paths generated by Focus are added on to the existing design constraints and the cumulative constraints are used to drive routing. Design size is not an issue with the exception generation flow - Focus has been run on designs containing 600K flops, 15M gates flat. The complexity of clocking is not an issue - Focus has been run on designs containing hundreds of clocks and generated clocks. Focus has been integrated with the leading chip-implementation tool flows. Adding Focus generated exceptions into the chip-implementation flow rarely has any impact on physical synthesis or routing runtime. QoR improves but runtimes stay unchanged.

Timing Exception Verification Flow

The Timing Exception Verification flow, shown in Figure 3, requires use of the Confirm product. The SDC file to be verified is written out from a static-timing or chip-implementation tool. The gate-level SDC file is mapped to RTL using Refocus. Refocus transforms gate-level object names to RTL names and reassembles flattened design hierarchy. The Refocus mapped SDC file along with the RTL for the design, .lib models for memories, simulation models for standard cells is provided as input to Confirm. Confirm formally verifies the false and multi-cycle paths in the SDC file.

When Confirm flags a timing exception as incorrect it also generates a report listing the timing paths constrained by the exception that are single cycle and contradict the user false or multi-cycle setting. This report may be viewed as a schematic using FishTail's integration with the Novas product family. Also, an assertion is generated that specifies why the false or

multi-cycle path definition is incorrect. Optionally, Confirm is able to re-write a partially correct timing exception when some of the paths constrained by the exception are truly false or multi-cycle while others are not. This allows users to specify exceptions in a loose manner and use Confirm to re-write the exception so that it accurately constraints the timing paths on a design.

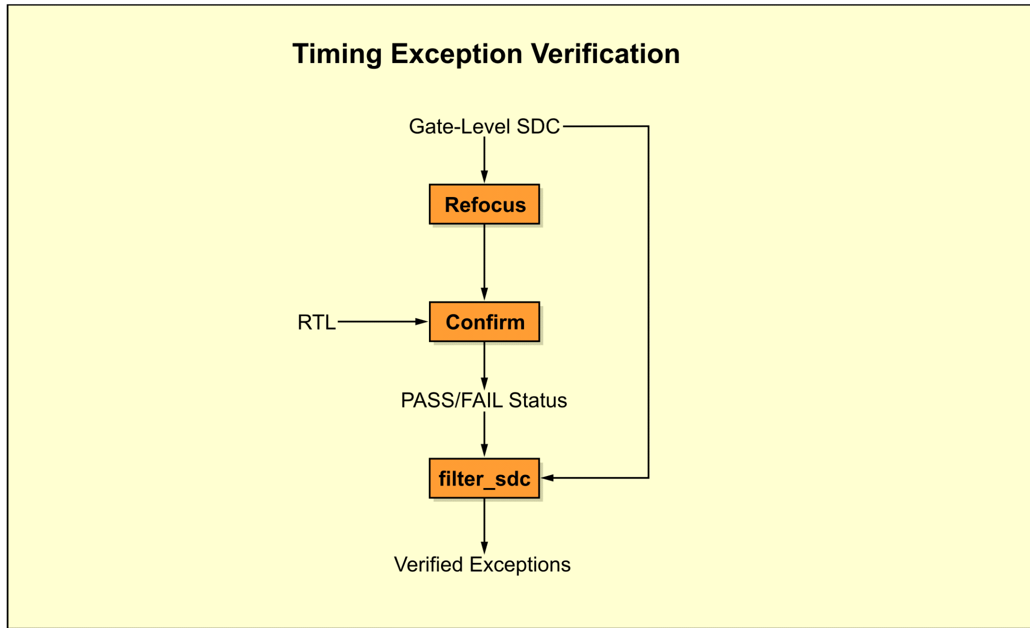


Figure 3: Timing Exception Verification Flow.

In addition to the formal verification of timing exceptions, Confirm also supports an assertion-based verification methodology. In this flow, Confirm generates assertions for each timing exception that describe a functional property that must be satisfied for the timing exception to be correct. The assertion file generated by Confirm is then verified using a functional simulator or using a formal property-checking tool. Assertions are generated in a variety of formats - PSL, SVA, OVA, OVL and Confirm has been integrated with the major functional simulators and property-checkers in the industry.

Constraint Management Flow

The Refocus product from FishTail addresses the constraint management needs of chip-implementation. The problem with today's implementation flow is that there are far too many versions of the design constraints for a project and these constraints are generated by far too many tools in the implementation flow. This leads to nervousness with regard to the correctness of the constraints used to finally tape-out a chip and also inefficiencies because the constraints used to implement the blocks on a chip cannot easily be reused when signing off on chip timing.

The chip-implementation flow promoted by FishTail, shown in Figure 4, requires maintaining a single version of the golden design constraints for a physical-synthesis block. Constraints are added and removed from this file as chip-implementation progresses either using automated tools like Focus or manual entry by design engineers. Each new entry to the

constraint file is verified using Confirm to ensure that the golden constraint file is always correct and that its contents have been formally proven.

As chip-implementation progresses Refocus is used to map the golden repository of design constraints to each new netlist version generated by different steps in the implementation flow (logic synthesis, DFT, physical synthesis, route, etc.). Refocus automatically handles name and hierarchy changes and can be provided register equivalences established by formal equivalence checking tools to aid in the constraint mapping. The key is that each tool in the implementation flow only changes the design netlist - not the design constraints. When block-implementation is complete and a full-chip netlist is assembled, the block golden constraints are promoted to the chip-level using Refocus. The promoted chip-level constraints are used during full-chip static signoff. During constraint promotion any inconsistencies between existing full-chip constraints and block constraints (in terms of clock and I/O delays) are reported.

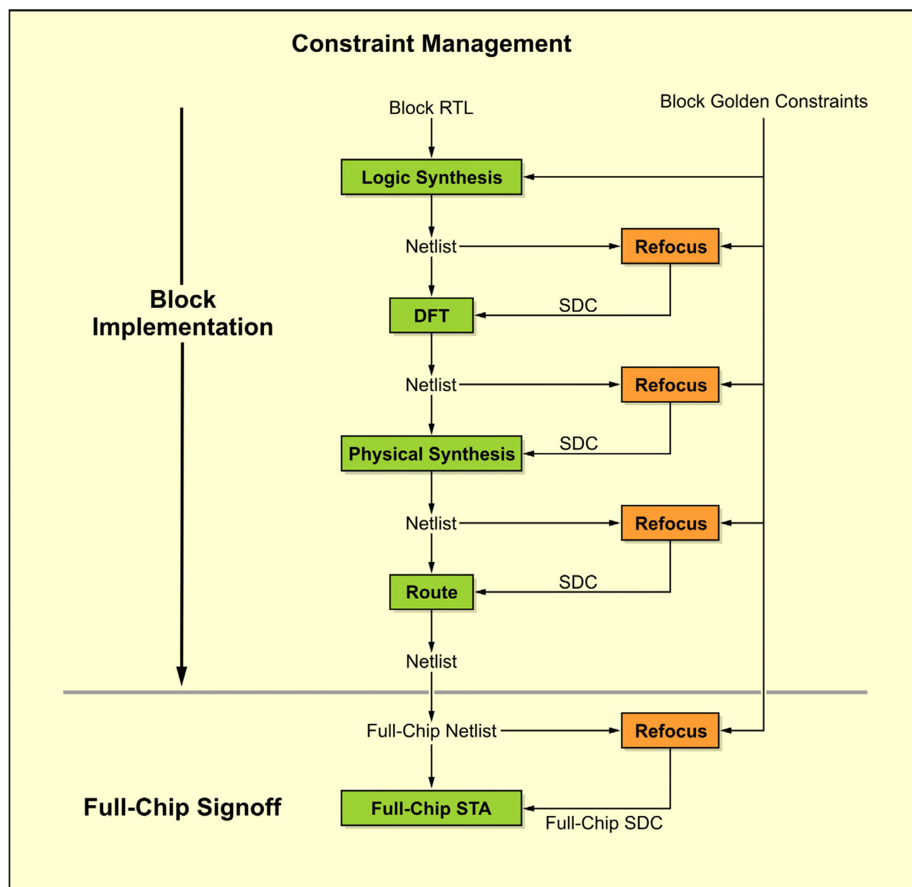


Figure 4: Constraint Management Flow.

Types of False Paths Formally Generated and Verified

There are four classes of false paths that Focus is able to generate and Confirm is able to verify:

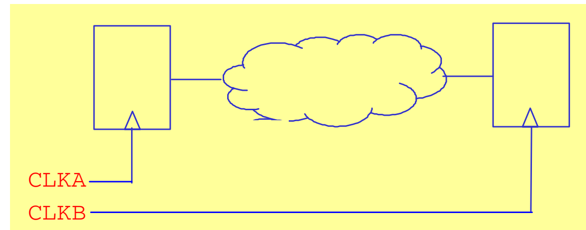
1) False paths resulting from asynchronous clock-domain crossings.

2) False paths resulting from clock modes.

3) False paths resulting from unsensitizable timing paths.

4) False paths resulting unreachable circuit states.

False Paths Resulting From Asynchronous Clock Domain Crossings



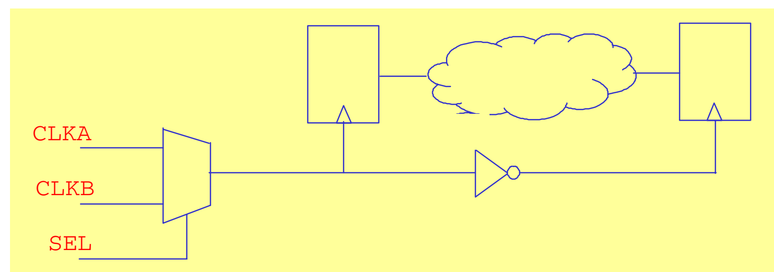

```
set_false_path -from [get_clocks clka] -to [get_clocks clkb]
```

Figure 5: False paths resulting from asynchronous clock-domain crossings.

When registers clocked by two asynchronous clocks communicate with each other, as shown in Figure 5, then a clock-to-clock false path between them is legitimate. By default, two clocks whose periods are equal or a multiple of two are regarded as synchronous to each other and otherwise they are treated as asynchronous to each other. So, a 3ns and 5ns clock would be regarded as asynchronous, while a 2ns and 4ns clock would be regarded as synchronous. Users can override the default grouping of clocks by adding the `-domain` option to the `create_clock` command. Two clocks with different `domain` values would be treated as asynchronous to each other regardless of their clock periods. So, for example, to make clka and clkb that both run at 5ns but are generated from different PLLs asynchronous to each other you do the following:

```
create_clock -period 5 clka -domain clka
create_clock -period 5 clkb -domain clkb
```

False Paths Resulting From Clock Modes

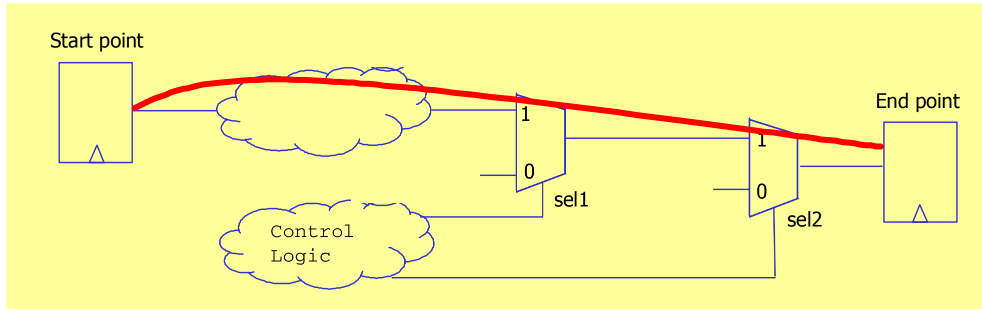



```
set_false_path -from [get_clocks clka] -to [get_clocks clkb]
```

Figure 6: False paths resulting from clock modes.

If two clocks are muxed in the clock-generation logic, as shown in Figure 6, so that only one of the two clocks is propagated to the registers on a design then it is legitimate to specify a clock-to-clock false path between them, because the clocks do not both simultaneously

propagate to the registers on a design. Setting such a clock-to-clock false path requires examining the clock-propagation characteristics across the entire design. If, for example, some registers on a design are always driven by a clock `clka`, and these registers communicate with other registers where `clka` and `clkb` are muxed, then it would be incorrect to set a clock-to-clock false path between `clka` and `clkb`.



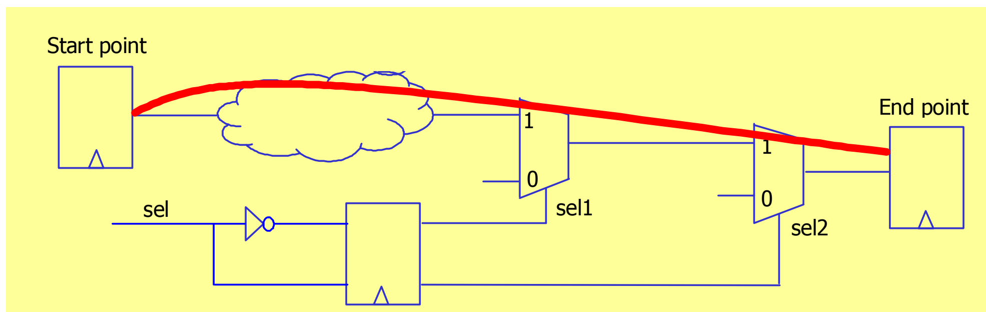
False Paths Resulting From Unsensitizable Paths

```
set_false_path -from [get_cells start_reg] -to [get_cells end_reg]
```

Figure 7: False paths resulting from unsensitizable paths.

If the combinational logic on the design is structured in a way such that regardless of register values or input port values some paths on a design are impossible to sensitize then it is legitimate to enter a false path in these situations. For example, in Figure 7, if the nets `sel1` and `sel2` are never high in the same clock-cycle then the path shown in red will be unsensitizable because its sensitization condition is `(sel1 & sel2)`. In this situation it is correct to specify a false path that constrains the unsensitizable paths. Paths that are combinationally false (i.e. unsensitizable for all register and input port values) are safe to apply even in test mode (for at-speed test) because these paths are false for all states of the design – even states that are not reachable during mission mode but are possibly reachable during test mode.

False Paths Resulting From Unreachable States




```
set_false_path -from [get_cells start_reg] -to [get_cells end_reg]
```

Figure 8: False paths resulting from unreachable states.

It is possible for paths to be false because of unreachable design states. For example, the path shown in red in Figure 8 is not combinationally false. If you ignore the values on the registers that drive `sel1` and `sel2` the path is sensitizable. However, once you consider how the values on these registers is computed sequentially then it is clear that the two registers will never take on the same value. As a result, the path shown in red is sequentially false. Focus generates 2-cycle multi-cycle paths for paths that are sequentially false. Confirm verifies sequential false paths entered by design engineers.

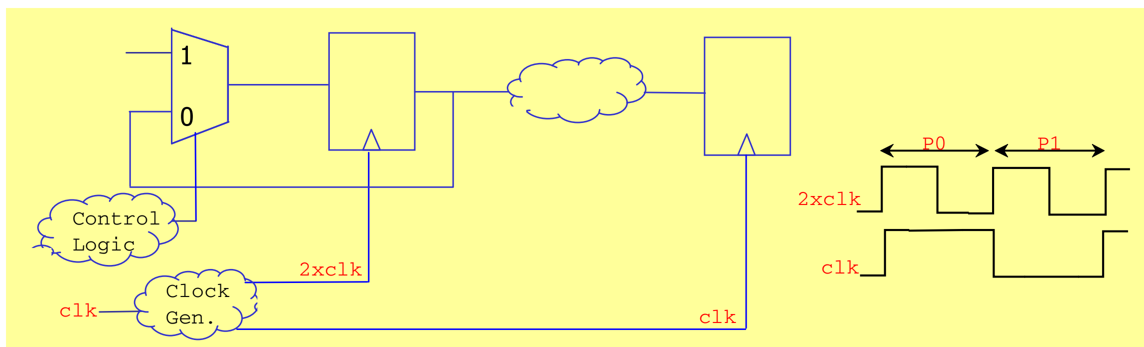
Types of Multi-Cycle Paths Formally Generated and Verified

There are two classes of multi-cycle paths that Focus is able to generate and Confirm is able to verify:

1) Multi-cycle paths between cross-clock domains.

2) Multi-cycle paths between same-clock domains.

Cross Clock Multi-Cycle Paths




```
set_multicycle_path -from [get_cells start_reg] \
                    -to [get_cells end_reg] -start -setup 2
set_multicycle_path -from [get_cells start_reg] \
                    -to [get_cells end_reg] -start -hold 1
```

Figure 9: Cross clock multi-cycle paths.

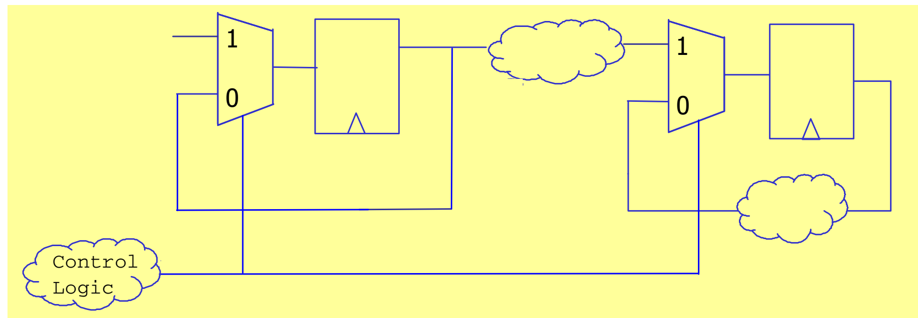
Cross-clock multi-cycle paths are legitimate when registers clocked by two synchronous clocks running at different speeds communicate with each other and, by intent, the design provides additional clock cycles for information to propagate from the launch register to the capture register. For example, in Figure 9, a register clocked by a 2x clock communicates with a register clocked by a 1x clock. Unless the RTL is designed to hold the value on the launch register for additional clock cycles, data generated by the fast clock will not be captured by the slow clock. However, simply holding the value on the launch clock for more than one cycle is also not sufficient for a multi-cycle path between fast and slow clocks. It is necessary that the launch register change value in phase `P0` only and never in phase `P1`.

Both Focus and Confirm formally establish if the design functionality is such that there are multi-cycle paths between registers clocked at different speeds (fast clock communicating with slow clock, or slow clock communicating with fast clock). Even though design functionality may allow for multi-cycle paths with more than two cycles of setup, Focus

generates two-cycle multi-cycle paths only and Confirm verifies whether at least two cycles of setup are available.

Same Clock Multi-Cycle Paths

It is possible to have multi-cycle paths between registers clocked by the same clock. For this to happen it must always be the case that when a launch register changes its value, the transition is not allowed to propagate to the capture register in the same clock cycle – instead the value propagates in a subsequent clock cycle. The design structure that allows same-clock multi-cycle paths is shown in Figure 10.




```
set_multicycle_path -from [get_cells start_reg] \
                    -to [get_cells end_reg] -end -setup 2
set_multicycle_path -from [get_cells start_reg] \
                    -to [get_cells end_reg] -end -hold 1
```

Figure 10: Same clock multi-cycle paths.

Focus and Confirm are both able to generate and verify same-clock multi-cycle paths.

Completeness of Focus

Focus discovers most of the timing exceptions on a design but not all of them. False paths that are not generated by Focus fall into one of two categories:

1) The false path applies to a timing don't care. For example, a false path from reset is often entered by designers because the reset timing is irrelevant. There is nothing functionally false about paths from the reset port on a design; it is just that it isn't necessary for these paths to meet timing.

2) The false path is from a static register that is software programmed to a constant value. As the value on a mode register does not change, engineers often mark as false all paths that start at such registers. This, again, is not something that can be functionally gleaned from the netlist or RTL.

Multi-cycle paths are not generated by Focus when an input port is used to control propagation along timing paths internal to a block. Focus assumes that an input port will assume a new value at each clock cycle. If, in fact, the input port has specific behavior (for example, being asserted every other clock cycle) that is the reason for multi-cycle behavior within a block, then in the absence of an appropriate clock waveform on the input port, Focus will not identify this multi-cycle behavior.

Correctness of Confirm

Confirm uses a separate approach to timing exception verification from that used by Focus to generate timing exceptions. The code used by Confirm is separate from Focus. FishTail guarantees that if there is an incorrect timing exception in an SDC file then Confirm will catch this mistake. Further, to facilitate third-party verification of timing exceptions Confirm supports an assertion-based verification methodology in addition to its formal verification capability. With this methodology, rather than formally prove the correctness of a timing exception Confirm can be asked to generate assertions that describe functional properties that must be satisfied for a timing exception to be correct. These assertions are then imported into a functional simulation or property checking tool and verified. Confirm has proven integration with all the major functional simulators and property checkers. Confirm can be asked to generate assertions in all the major assertion languages (PSL, OVL, OVA, SVA).

Summary

FishTail products are mature and have been deployed by leading semiconductor firms to tape-out chips. Exception generation from Focus reduces the number of timing violations after place-and-route by 25-75%, saving weeks in chip-implementation TAT. Exception verification from Confirm has routinely caught mistakes made by engineers when they entered timing exceptions manually. If you are signing off a chip that has timing exceptions entered by engineers you are playing with fire if you allow tapeout to proceed without first formally verifying these timing exceptions. It is irrelevant that these exceptions have been used on prior tapeouts have you ensured that the exceptions have kept up-to date with design changes made since the last tapeout? Finally, Refocus allows you to smoothly manage your constraints throughout the entire chip-implementation and signoff flow.