Static Timing Fundamentals

Jason Rziha

Microchip Technology
Chandler, AZ USA

http://www.microchip.com

ABSTRACT

This paper examines the fundamentals of Static Timing and applies them to developing and debugging timing constraints. Almost all difficulties users encounter with timing constraints can be solved by applying the fundamental rules of STA and combining that with an understanding of how the tools interpret them. Real world problems solved by applying and interpreting the fundamentals are used for illustration.

Table of Contents

Table of Figures

1. Introduction

Modern SOC implementation flows all depend on static timing analysis to enable the tools to optimize the design. However, many users have an inadequate understanding of the fundamentals of static timing anlaysis. This leads to increased cycle time for developing timing constraints, applying them to the implementation flow, and debugging issues as they arise.

While users often spend significant amounts of time pursuing tool specific analysis, in practically every instance, the fastest path to resolving the problem involves applying the fundamental rules of timing analysis. Regardless of tool or vendor used, the fundamentals of static timing remain the same. So whether one uses Design Compiler, PrimeTime, or IC Compiler applying these fundamentals will accelerate the debug cycle and help achieve timing closure.

2. Timing Analysis Fundamentals

2.1 Fundamental #1: Graph Based Approach

At its simplest level, static timing analysis is done by reducing the netlist to a series of nodes and arcs to form a graph. The timing engines don't care about the functionality of the cells- they simply add the delays along the arcs to an endpoint for a launch and capture path, then compare the two values for the desired relationship (launch arriving before capture for setupcheks, capture arriving before launch for hold checks). While the actual graph is more involved – keeping earliest and latest arrivals, and for both rising and falling edges – the fundamental, simple, model illustrated in is useful for debugging many situations.

2.1.1 Translating Gates to the Graph

Examples of individual cell representations, with the input net delays shown as black arcs and the internal cell delays shown as red arcs:

2 Input OR Gate:

2 Input AND Gate:

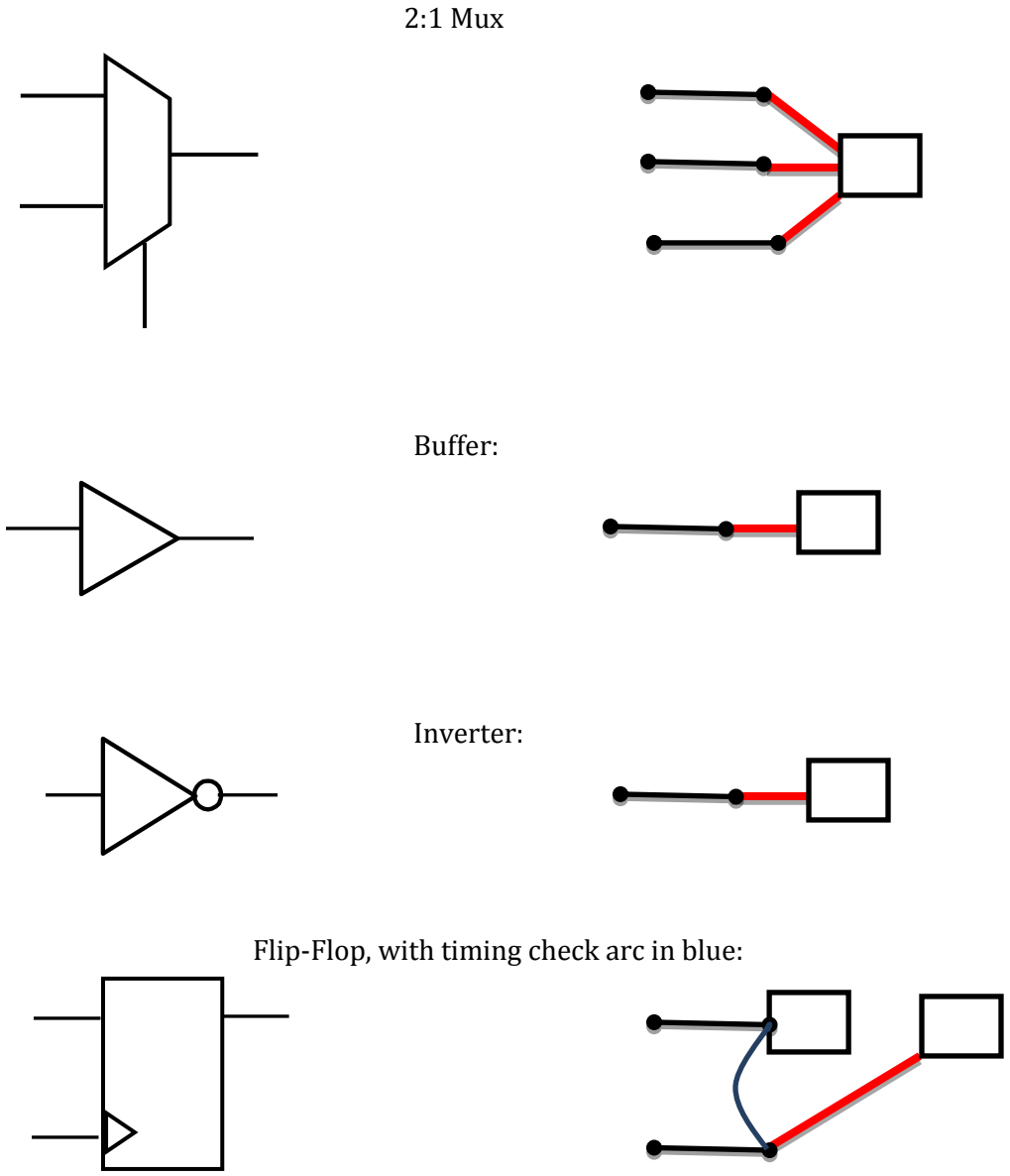

Figure 1: How common cells are viewed as part of the timing graph

Note that in many cases, the graphical representation of different cells (buffers and inverters, or two input AND/OR gates for example) are identical.

2.1.2 Assembling the Path and Calculating the Slack

And several gates assembled into a timing path: (Note that the knowledge of the cell's functionality has been lost).

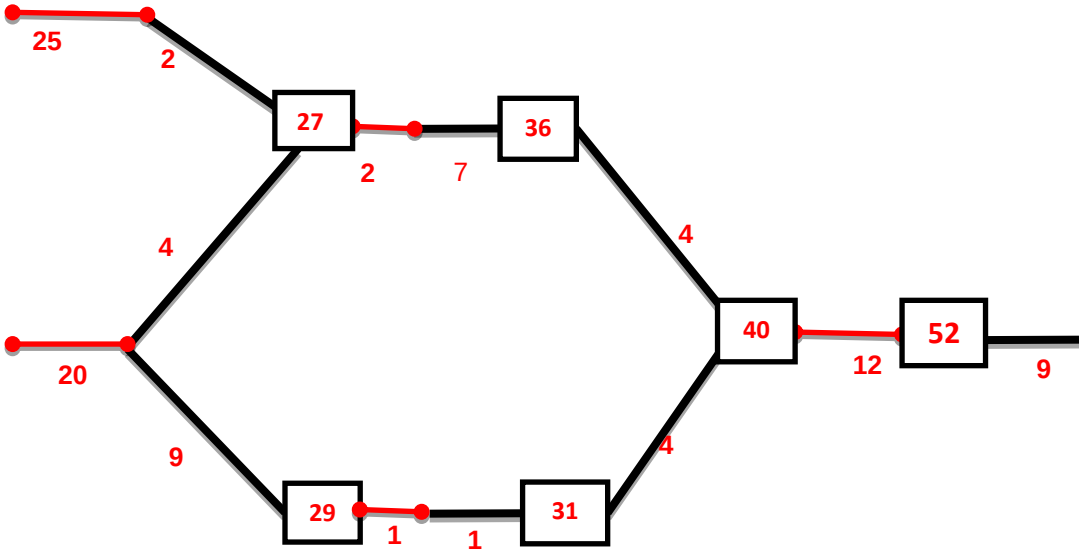

Figure 2: Graphical representation of a timing path

Finally, the graphical representation of a timing check with both its launch and capture paths shown, tied together with the setup check arc in the flip-flop.

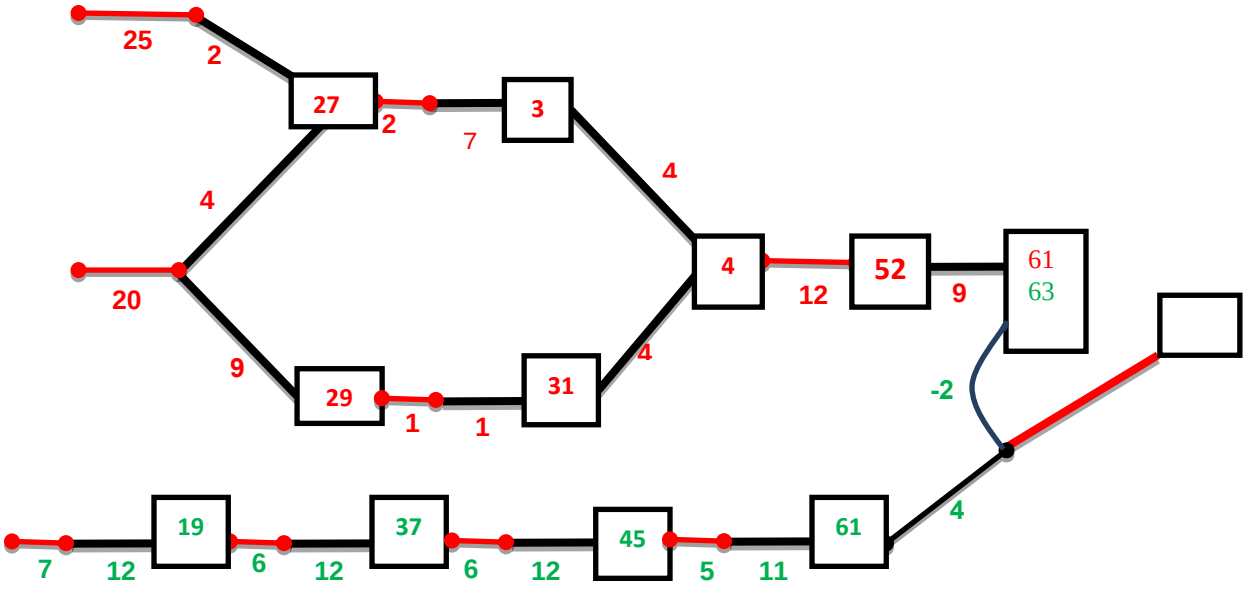

Figure 3:Graphical representation of launch and capture paths

2.1.3 Impact of the Graph Model

The impact of this model is perhaps most significant in understanding the difference between the messages “No Paths” and “No Constrained Paths”. In the former case, the tool is stating that there is not a launch path on the graph that meets the requirements the user specified. In the latter case, it is possible to traverse the graph, however upon arrival at the endpoint there is not any required time to compare our arrival against. Where the user would investigate for problems and the approaches used will vary considerably depending on which message is provided by the tool .

For example, ifset_disable_timing were to remove the arc with a 12 unit delay near the end of this path, we’d end up with the graph below:

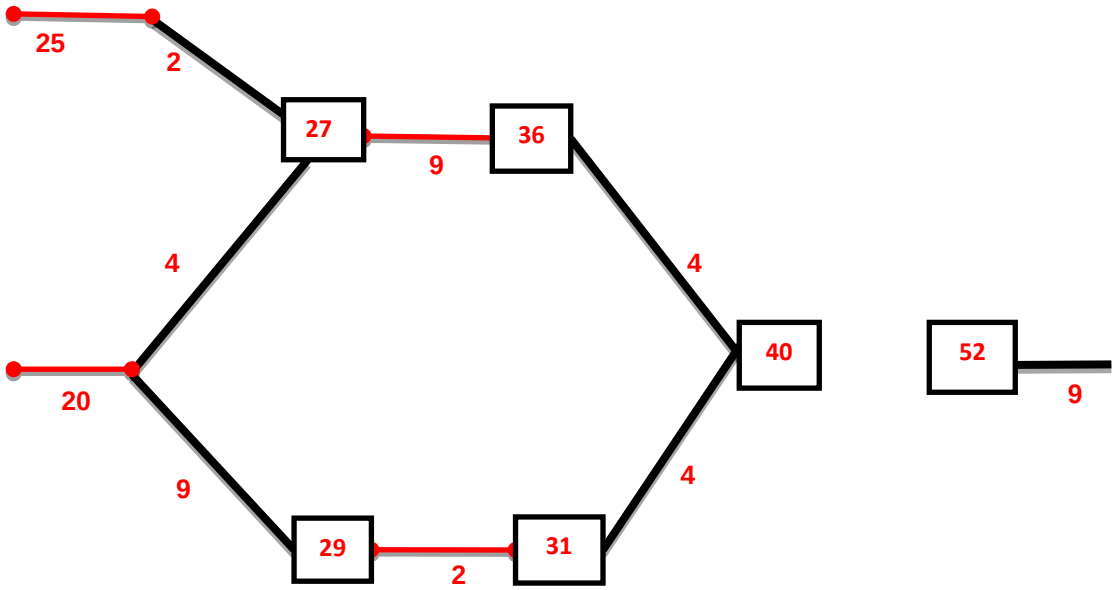

Figure 4: Effect on the graph of disabling a timing arc

The result of this change would be that if one asked the tool to report timing to the endpoint, and if this is the launch path, the response would be “No Paths”. However, if this was the capture path the tool would respond with “No Constrained paths” as it was unable to calculate a required time.Understanding that difference helps the user focus their debugging efforts in the appropriate location.

This model is also useful as a way to visualize the impact of certain commands, and to understand why the tools don’t disable arcs based on functionality. For example , using theset_disable_timing commandto remove a cell or net delay is equivalent to taking one of the arcs in the model above and removing it, possibly resulting in a “No Paths” message. Removing the timing check arc at the end of the path would result in “No constrained paths”. The effects of set_case_analysis are similar. However, if the path traverses a mux that would only be enabled when a register was 0 and then reaches an AND gate whose other path is tied to the register output, the tool wouldn’t disable the path through the AND gate despite it being impossible for this path to occur in the design.

2.1.4 Tools for Tracing the Graph

Understanding how the tool models the paths and interpreting the messages "No Paths" and "No Constrained Paths" helps the user narrow their search for a problem. Once they've decided where to look, how do they trace the path to find the location of the problem? Manually tracing the path either through the Verilog netlist or schematic are not only painfully slow and error prone, they also will not reveal problems in the path caused by disabled timing arcs. Fortunately the Synopsys toolset provides some useful commands for tracing timing paths – `all_fanin` and `all_fanout`.

All_fanin works by tracing timing paths backward from the target pin, passing through hierarchy, stopping at:

→ Input ports

→ CLK flop pins, G and D latch pins

→ Cell output pins where all timing arcs are disabled or missing

All_fanout works by tracing timing paths forward from the target pin, passing through hierarchy, stopping at:

→Output ports

→Input flop and latch pins

→Cell input pins where all timing arcs are disabled or missing

These commands are perfect for tracing timing paths and determining if we can walk the graph to a point of interest.

2.2 Fundamental #2: Every Timing Path is the Same

Every timing path is modeled the same way – a launch flip flop connected to a combinational cloud that feeds into the capture flip flop.

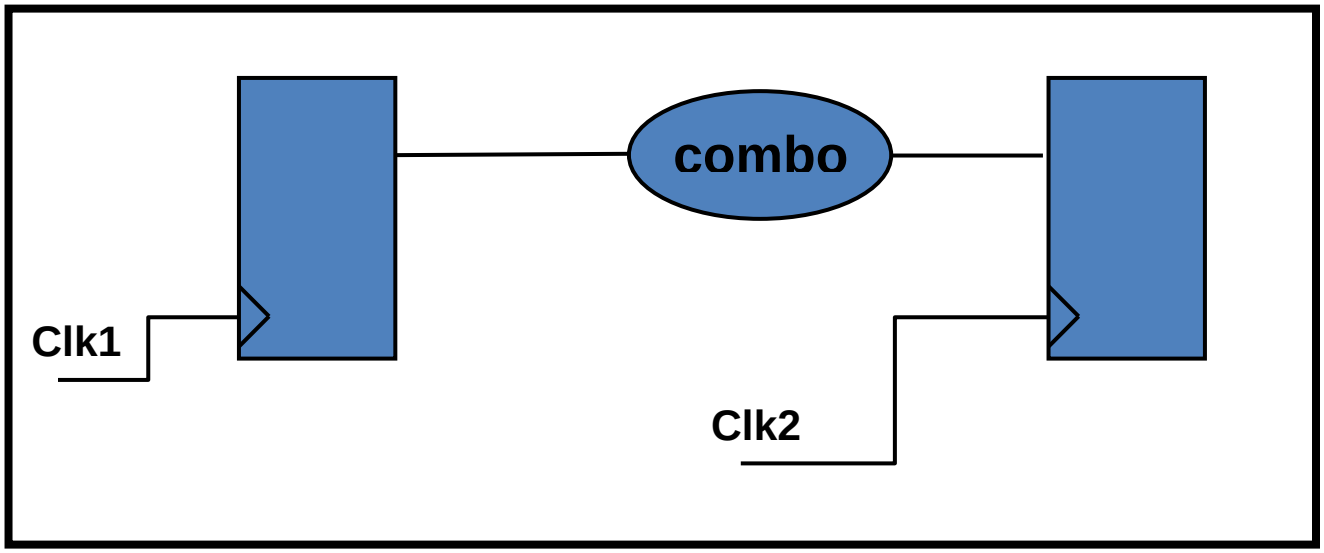

Figure 5: Example Timing Path

This is true even if the timing path includes inputs– the external part of the timing path simply becomes virtual.

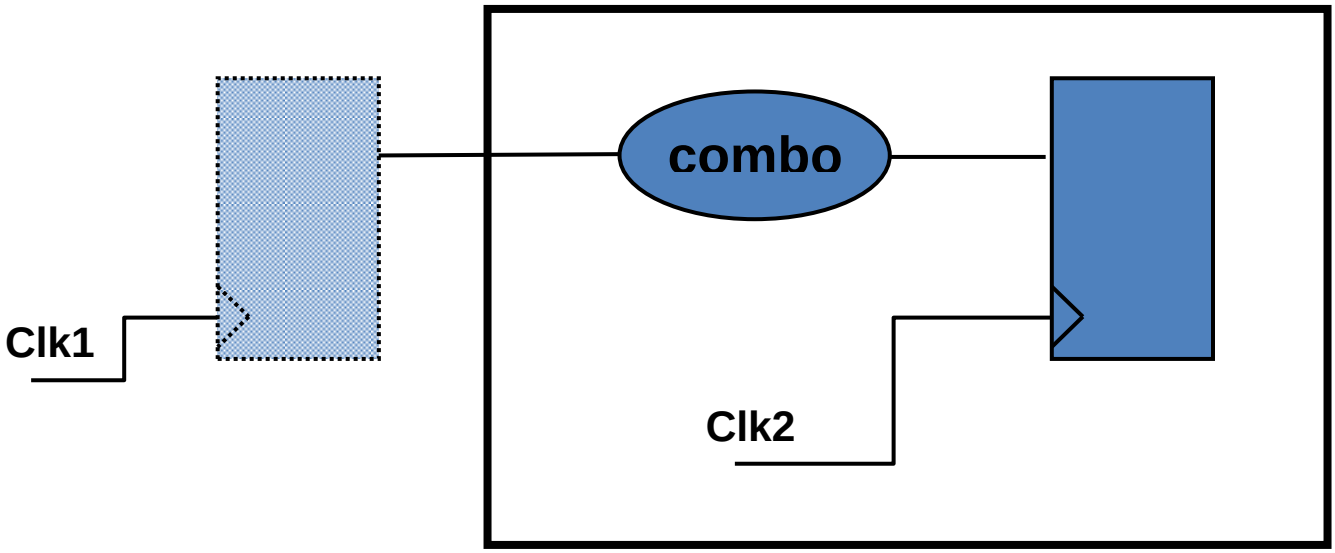

Figure 6: Example Input Path Timing Path

Likewise for paths that include outputs – the part of the path outside of the design becomes virtual.

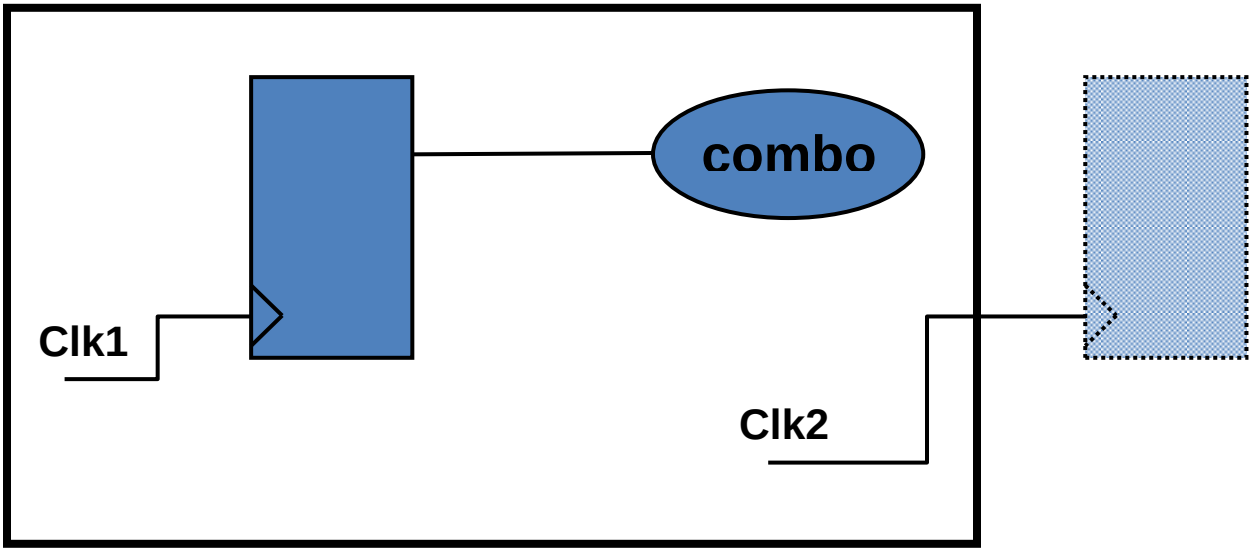

Figure 7: Example Output Path Timing Path

For paths that are purely combinational within the design, the launch and capture flops are both virtual- but they're still there

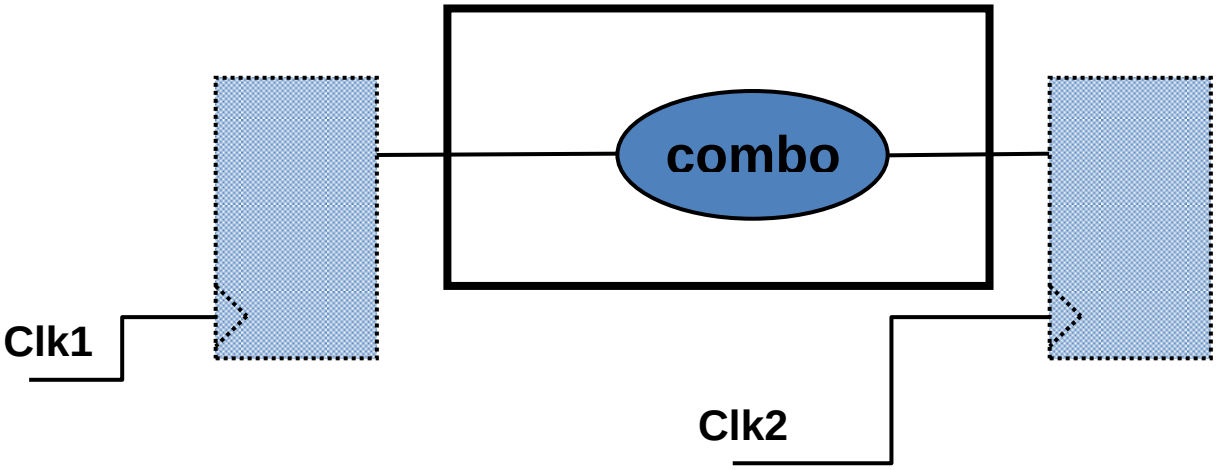

Figure 8: Example Combinational Feedthrough Timing Path

Modeling paths in this way helps users particularly understand what is being described with `set_input_delay`and `set_output delay`, including why they have to specify a clock for those commands.

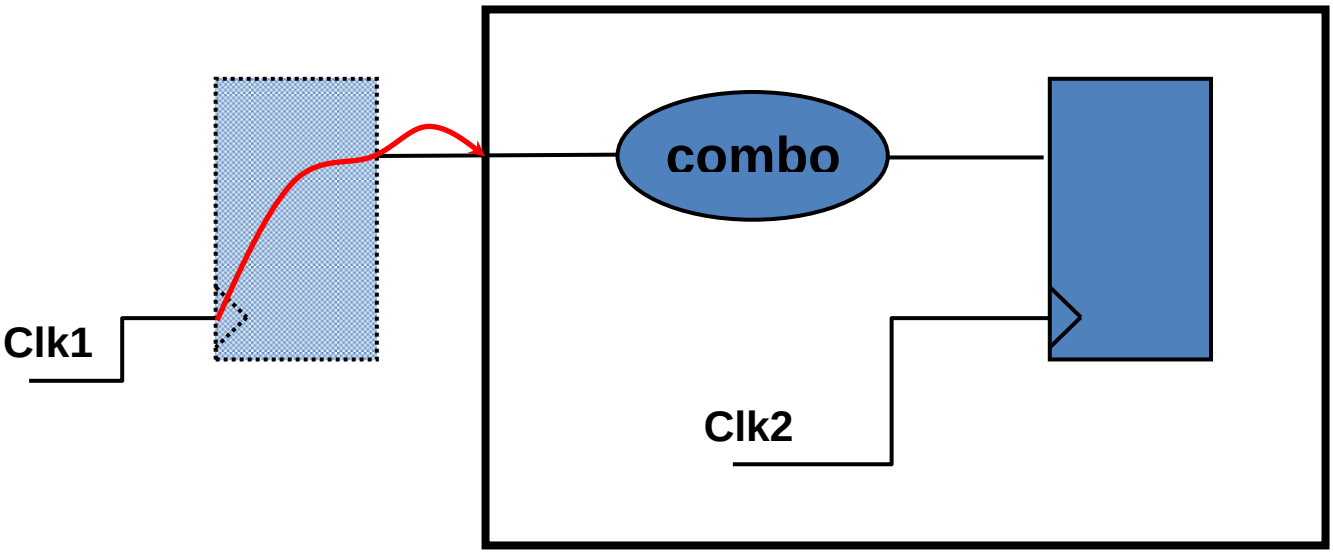

Figure 9: Portion of Path Delay Represented by set_input_delay

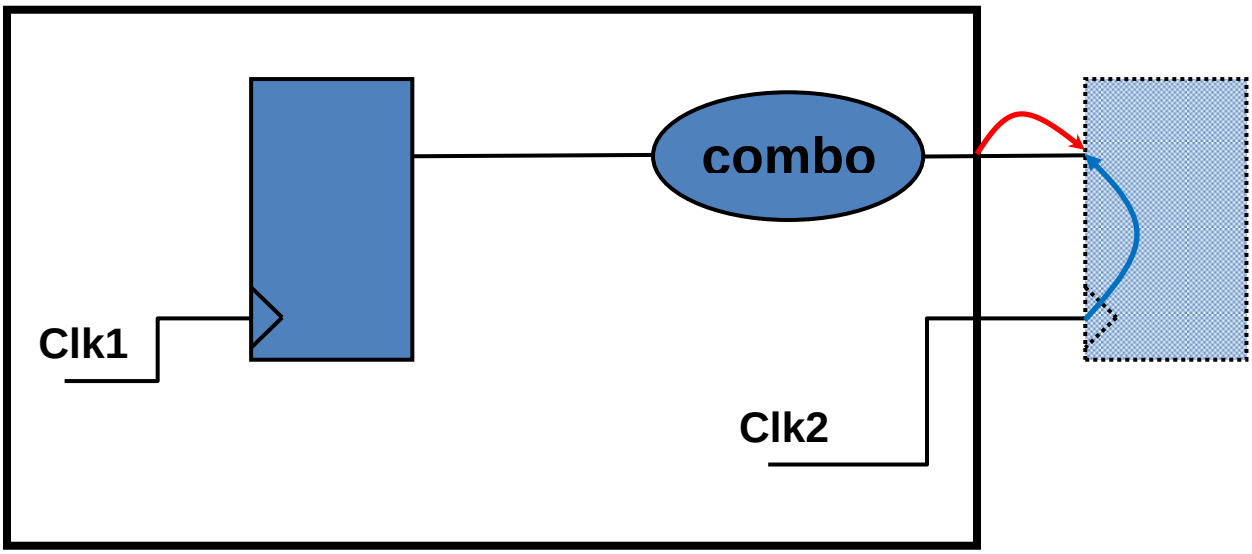

Figure 10: Portion of PathDelay Represented by set_output_delay

2.3 Fundamental #3: Startpoints and Endpoints

2.3.1 Textbook Definition

If, during a job interview, you asked a candidate engineer about the startpoints of a timing path, you're likely to receive (at best) the textbook answer:

-primary inputs

-clock inputs of a sequential element (although they'll usually say Flip-Flop)

Likewise for endpoints, you'll probably get:

-primary outputs

-D pins of flops

With the savvier engineers adding in:

-enable pins of clock gating cells

Although there's also a good chance (around 50% in the author's experience) that you'll get something that's at least partially wrong. The literal textbook mistake here is to use the output pin of a flip-flop as the startpoint in your constraint definitions- resulting in an error and an ignored constraint. -The start/end points of timing paths are so commonly misunderstood that incorrect examples have even been seen at Synopsys SNUG presentations presented by Synopsys employees.

2.3.2 Startpoints and Endpoints – what the tools support

While the textbook answer is great, it's not a complete picture of what the tools actually support. The tools support a much more robust set of start and endpoints- the proper application of which can be used to solve a number of constraint problems.

The timing pathstartpoints supported by Synopsys tools are:

-primary input pins (including inouts)

-clock pins of sequential elements

-sequential elements

-D input of a latch

-clock objects

-clock gating cell clock pin

-any pin with an input delay specified

The timing path endpoints supported by Synopsys tools are:

-primary outputs (or inouts)

-Data inputs of sequential elements

-sequential elements

-clock object

-clock gate data pin

-any pin with an output delay specified

3. Applying the Fundamentals to Real World Examples.

3.1 What is the tool doing?

The first example illustrates how the fundamentals can be used to understand what a tool is trying to do. In this case, IC Compiler was reporting that the design was timing clean, but on bringing back it into PrimeTime, PrimeTime was complaining about numerous hold violations- many of which were 1000ns or more.

This example is trivial to solve several ways, (a quick report_timing is sufficient), but applying one's knowledge about the start/end points supported by the tool makes even that unnecessary.Inspecting the report generated by `report_constraint –all_violators` revealed a list like the one below:

…

chip/block/block2/and3

chip/block/block3/or2

chip/block/block2/or3

chip/block/block4/and2

chip/block/block2/or3

…

Simply examining the list revealed that the listed endpoints were all and/or gates. Applying the 3rd fundamental – endpoints supported by the tool, one is quickly able to eliminate all possibilities except clock gate data pins. Since these were not integrated clock gates with library defined gating checks, it was further possible to deduce that PrimeTime had inferred the clock gating checks. Analysis showed that this was correct and a quick serach on SolvNet retuned the scripts to find and disable the improperly inferred clock gating checks, eliminating the errors and allowing the user to focus on real issues.[iii]

When two tools give different answers it can be difficult to determine whether they're simply making different assumptions, or whether one has a bug. Applying the fundamentals of STA made it possible to understand that, while inconsistent with IC Compiler, what PrimeTime was doing was expected (albeit incorrect in this case) behavior, allowing the user to quickly adjust the tool settings to properly time their design.

3.2 Conditional Multi-Cycle

This example illustrates the failure of the textbook definition of start/end points. -A user wanted to define a path between two registers as multicycle, but only when launched by a particular clock. When launched by other clocks it should be single cycle. The logic cloud between the registers was the same regardless of launch clock, so one couldn't use a –through switch to handle the selection.

Simply applying the textbook definition of valid startpoints will lead one to the conclusion that this can't be done. You can't conditionally declare a multicycle path starting at the clock pin of a register.

```
set_multicycle_path –from [get_pin<launch_register_clock_pin>] \
      –to [get_pin<capture_register_data_pin>]
```

This command will catch all the paths through this path- even those intended to be single cycle.

The solution lies in the larger list of supported start/end points of the tools. Using the clock object (instead of the register clock pin) as the startpointallows one to create the desired constraint.

```
set_multicycle_path –from [get_clock<launch_clock>] -through
[get_pinlaunch_flop/Q] \
      -to [get_pin<capture_register_data_pin>]
```

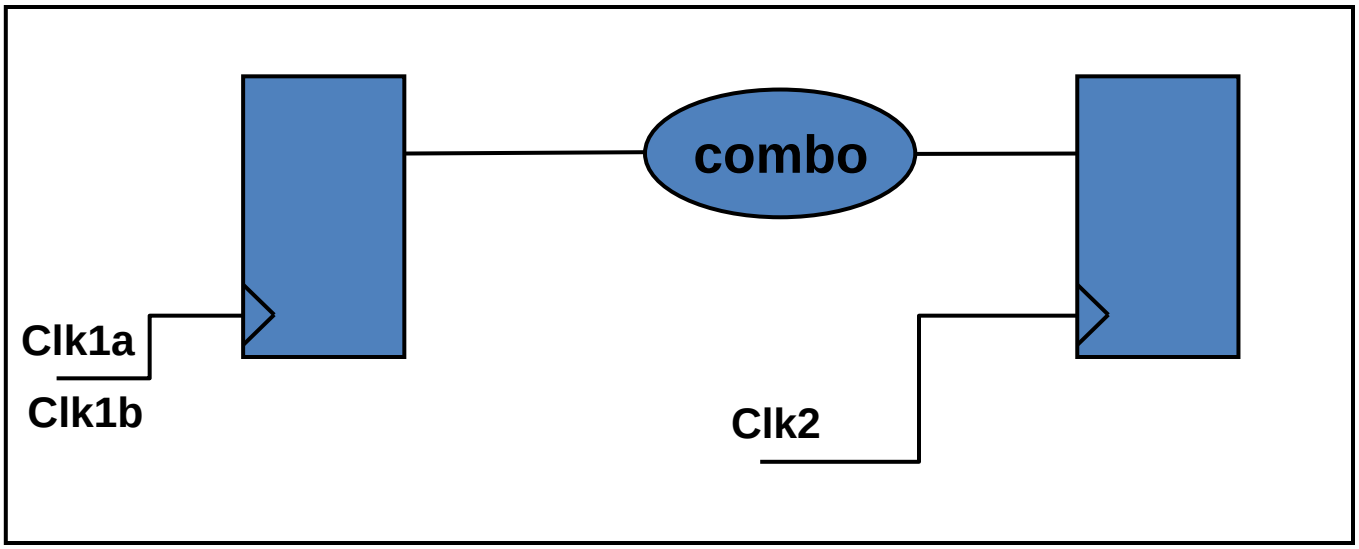

Figure 11: Conditional Multicycle Path schematic

3.3 Does this path exist or not?

Here a user was analyzing a path in IC-Compiler that resembled the circuit shown in Figure 12Figure 12: ICC said this path didn't exist when clocked by a specific clock where the register's output fed back into the enable of the clock gate on its clock. PrimeTime properly reported this path, but IC Compiler would report it as "No Paths" after constraints were applied.

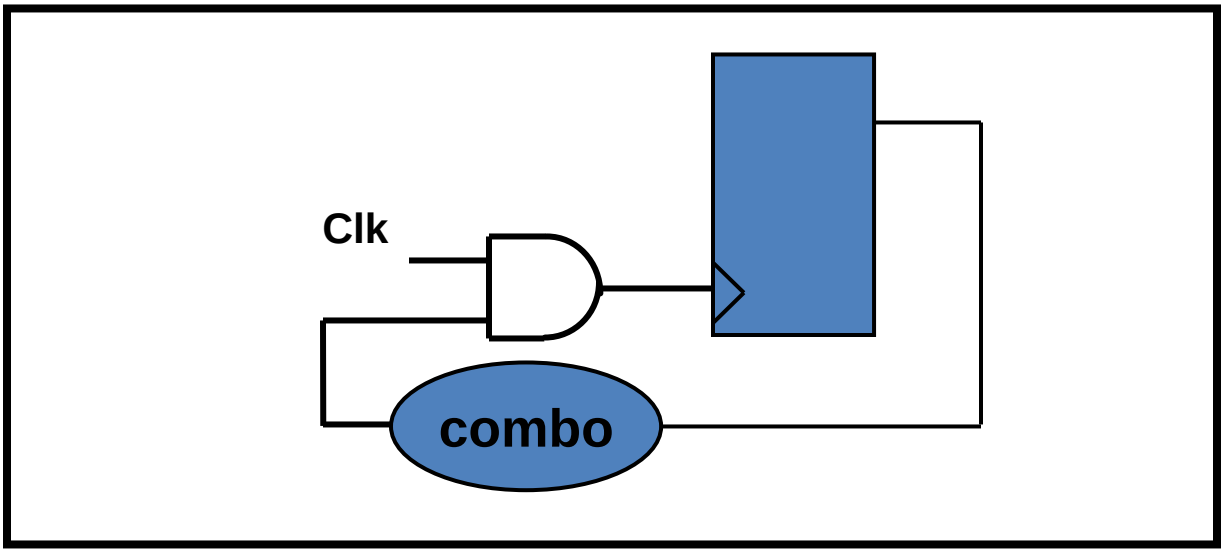

Figure 12: ICC said this path didn't exist when clocked by a specific clock.

With two tools making different claims about a path, what should one do? Get rid of one of the tools – in this case, the one we thought was right- PrimeTime. This eliminated any fingerpointing between tools.

Then we set about verifying our belief that IC Compiler's claim of "No Paths" was incorrect. To show that a path did exist, we needed a way to demonstrate walking it. Simply examining the schematic wouldn't be sufficient – one would also have to verify that none of the timing arcs in that schematic were disabled. While this would be possible to do, it would be tedious.

Fortunatelyall_fanin/all_fanout essentially work by walking the timing graph either forward or backward until a start or endpoint is reached, or the path is blocked by a missing timing arc. If "`all_fanin -start -flat -to <endpoint>`" showed the-Q pin of the register then we knew we had a path between those points. Likewise, if and "`all_fanout -end -flat -from <FF/Q>`"[1]showed the enable pin of the clock gate, we knew we had a path. Using both commands gave two ways to prove that there was a valid path between the start/end points. The final checks done were to demonstrate that the clock could reach the other pin of the gating cell, verifying that that portion of the path existed.

[1] These techniquese for tracing timing paths are discussed extensively in the "PrimeTime2: Debugging Constraints" class offered by Synopsys.

Because we did all the analysis within a single tool, there was no argument about which tool was wrong. -One way or another, IC Compiler was incorrect- either the report_timing was improperly saying "No Paths" or all_fanin and all_fanout were reporting a path that didn't exist. In Example #1, the fundamentals were used to quickly show that the tool was correct, while here they're used almost as quickly to provide a high level of confidence that the tool was incorrect.

3.4 Timing Self -Timed Logic

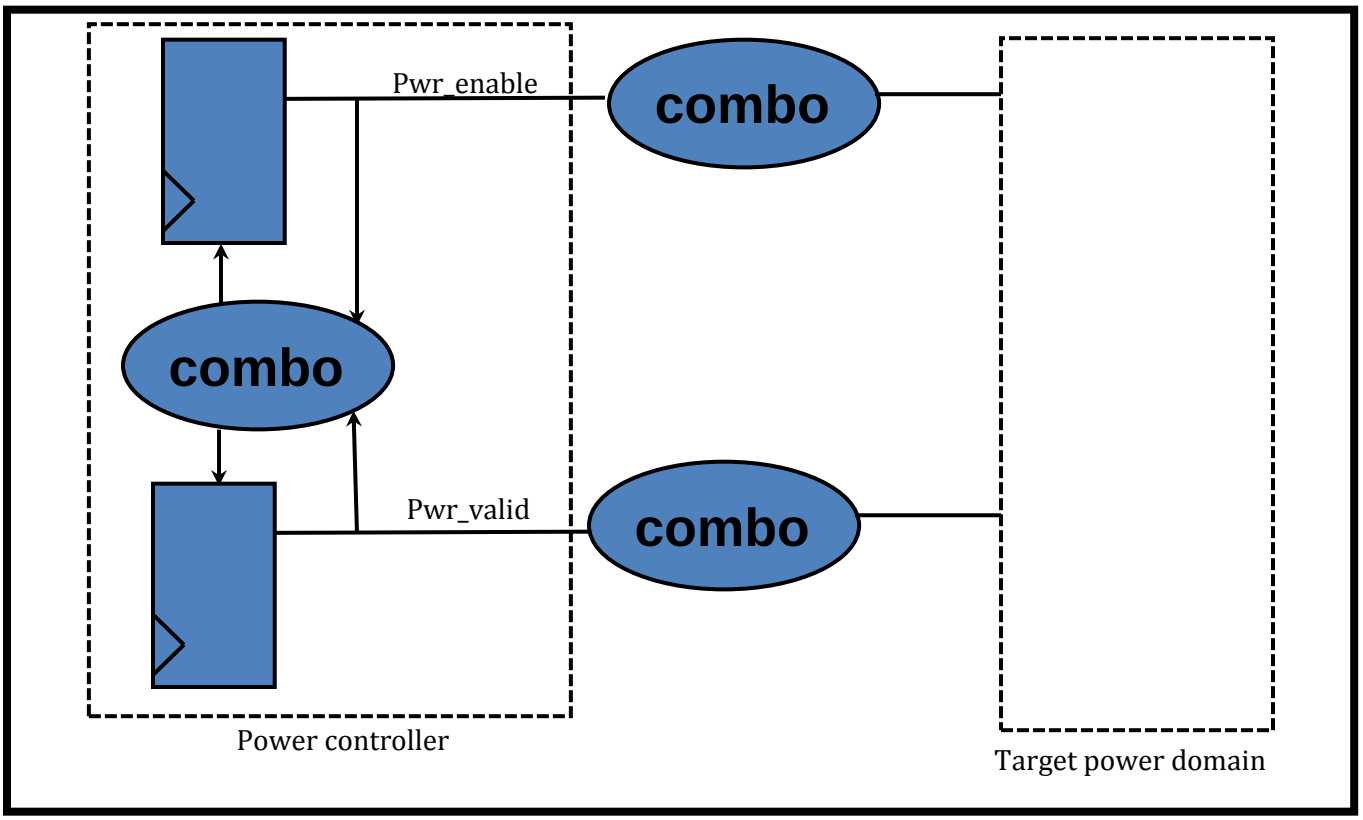

Figure 13: Self-Timed Logic

Here the user built a power controller using self-timed logic that fed back into the set and reset ports of the controlling registers, rendering the generation of the signals of interest totally asynchronous. However, the user wanted to guarantee that the pwr_enable signal went away after pwr_valid when powering down, and that pwr_enable came back before pwr_valid when powering up. The design ensured that the signals had the proper relationship when they left the controller block, but the user wanted to test the phase relationship at the destination and ensure that the required relationship was maintained.

As described, this design could not be analyzed with static timing techniques. Because the paths are asynchronous, we can't compare the arrival times at the target domain to ensure that they have the proper phase relationship. Instead of trying to analyze the asynchrous paths, the solution here lies

in observing that the pwr_enable and pwr_valid signals do have a defined phase relationship when they exit the power controller. That, plus the controller block being hardened, lets us change the model of the controller from the asynchronous model above to something that obeys fundamental #2.

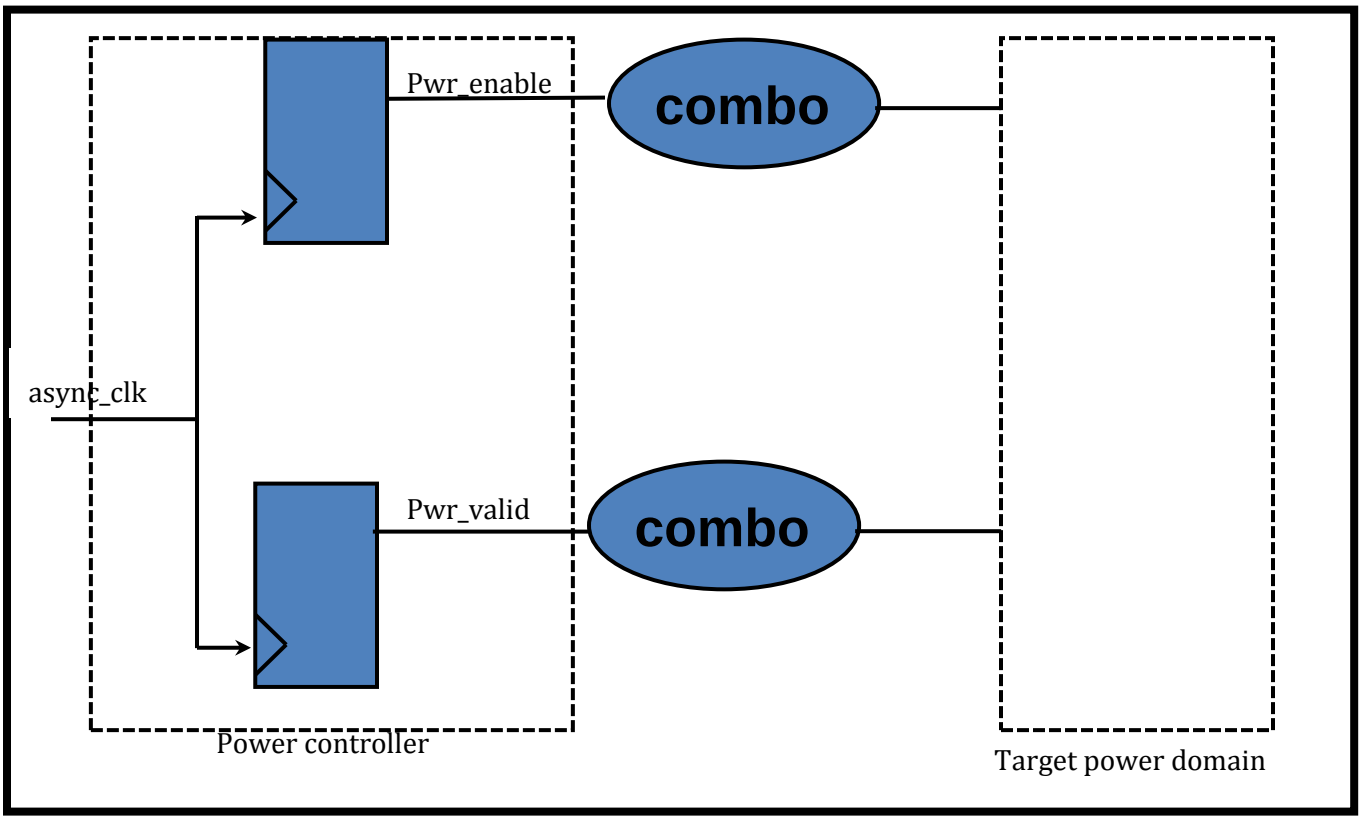

Figure 14: Restructured Self- Timed Circuit

By changing the model of the power controller, we can represent the fact that while the pwr_enable and pwr_valid signals are asynchronous to the rest of the design, they have a defined phase relationship between them. Once we have a model for the design that follows the fundamentals of static timing, ensuring the proper phase relationship at the endpoints is simple using a data-to-data check:

```
set_clock_groups –asynchronous –group [get_clockasync_clock] \
  -group [ remove_from_collection [all_clocks] [get_clockasync_clock]
set_data_check –setup –from [get_pin<pwr_valid pin> ] –to \
 [get_pin<pwr_enable_pin>]  <value>
set_data_check –hold –from [get_pin<pwr_valid pin> ] –to \
 [get_pin<pwr_enable_pin>]  <value>
```

3.5 Externally clocked interface

This is example which many will (rightfully) consider basic. However, it provides an example of how not applying the fundamentals can prevent even experienced users from getting the desired information from the tools.

An external interface was clocked externally, with data driven on one edge of the clock and captured on the following, opposite phase, edge. Two sets of registers capture on opposite phases of the external clock.

1. Rising edge of EXTCLK, DATA changes, falling edge of EXTCLK, DATA is captured in CFF1 register. Max delay
2. Falling edge of EXTCLK, DATA changes, rising edge of EXTCLK hold for LFF. Min delay

3. Falling edge of EXTCLK, DATA changes, rising edge of EXTCLK , DATA is captured in CFF2 register. Max delay
4. Rising edge of EXTCLK, DATA changes, falling edge of EXTCLK hold for LFF. Min delay

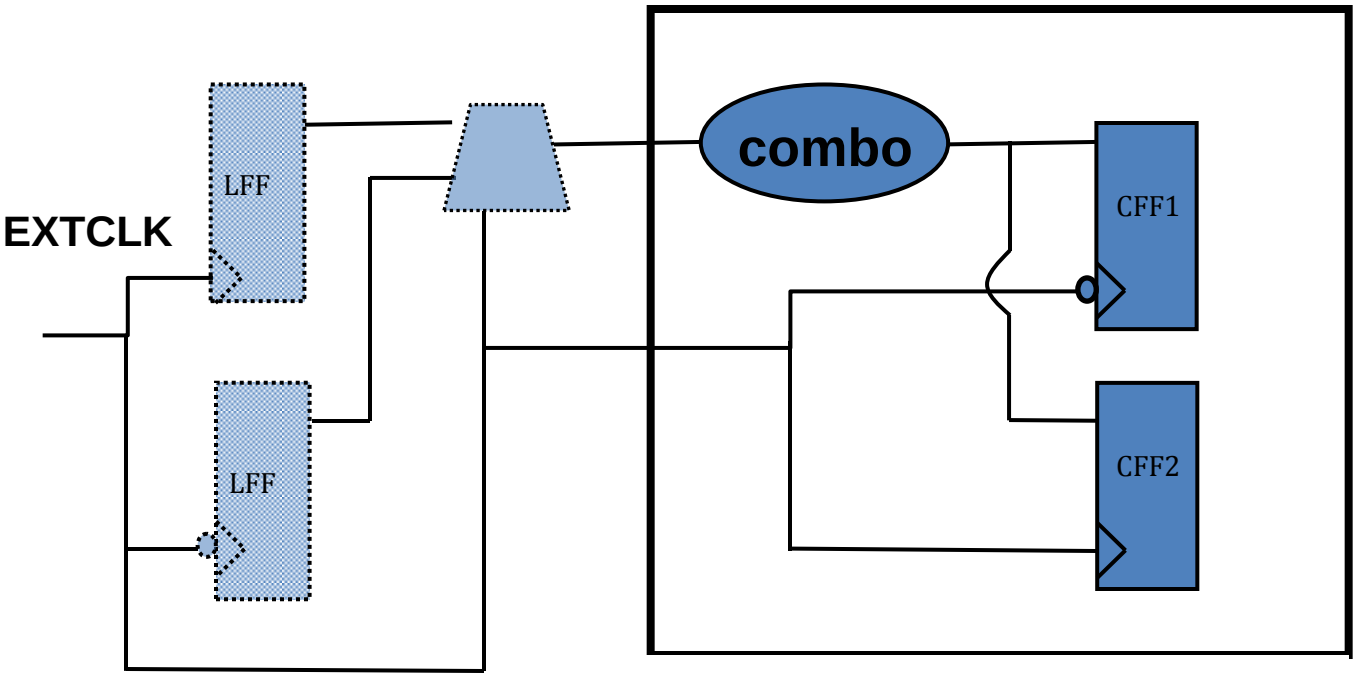

Figure 15: Schematic of Circuit

The (abbreviated) user defined constraints for the EXTCLK and DATA input delays are:

```
create_clock -period  $PERIOD -name EXTCLK [get_ports EXTCLK]
set_clock_uncertainty -setup $SETUP_UNCERT [get_clocks EXTCLK]
```

```
set_clock_uncertainty -hold $HOLD_UNCERT [get_clocks EXTCLK]
set_clock_transition 0.5 [get_clocks EXTCLK]

set_input_delay -min $INPUT_MIN -clock TXCLK156 [get_pin DATA]
set_input_delay -max $INPUT_MAX -clock TXCLK156 [get_pinDATA ]
set_input_delay –min $INPUT_MIN -clock_fall -clock EXTCLK \
    -add [get_pinDATA]
set_input_delay -max $INPUT_MAX -clock_fall -clock EXTCLK \
    –add [get_pinDATA]
```

When the user attempts to get the timing reports using:

```
report_timing -from DATA[*] –to[get_pinchip/hier/reg1_*_/D] –cap \
   -path full_clock_expanded
report_timing -from DATA[*] -to [get_pin chip/hier/reg1_*_/D] –cap \
 -path full_clock_expanded
report_timing -from DATA[*] -to [get_pin chip/hier/reg2_*_/D] -cap \
   -pathfull_clock_expanded–delay_type min
report_timing -from DATA[*] -to [get_pin chip/hier/reg2_*_/D –cap \
-pathfull_clock_expanded–delay_type min
```

They're left disappointed with the results and wonder why the tooldidn't report the timing for all of the launch/capture edge pairs they wanted.

3.5.1 What Went Wrong

Thedisappointing results were caused by two things: Not properly understanding the path being analyzed, and only using the textbook definition of the start/end points. Figure 16**Error! Reference source not found.**shows basically what the user envisioned:

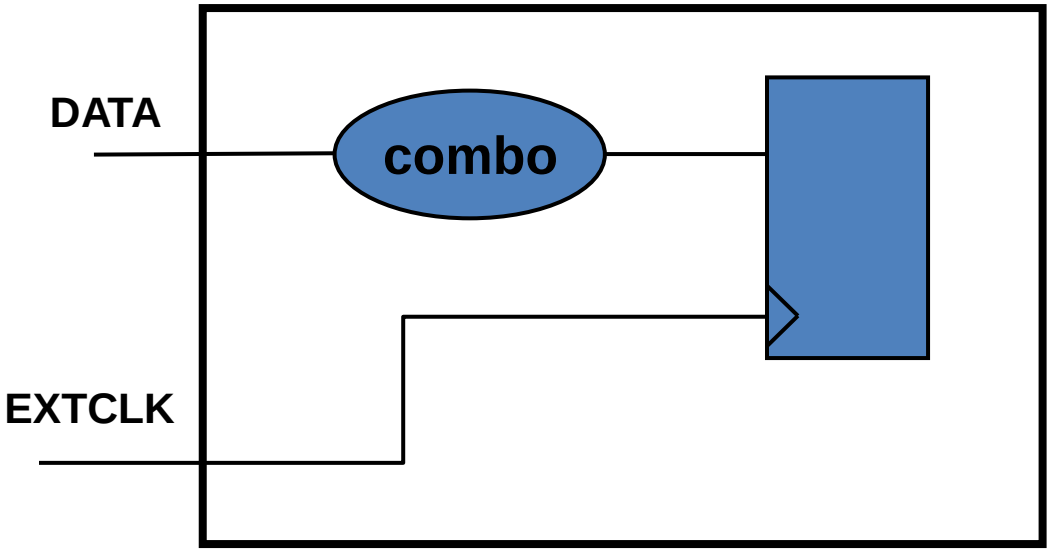

Figure 16: What the User Envisioned

Overlooking the virtual external register made it impossible to select the path based on the edge of the launch clock, and then they applied the textbook start/end points, trying to time from the input port to the D pin of the register.

3.5.1 Applying the Fundamentals

The solution here is to apply fundamentals #2 and #3 together. We'll extend the simple model of fundamental #2 to account for the clocking on opposite phases, resulting in the simplified circuit shown in Figure 17.

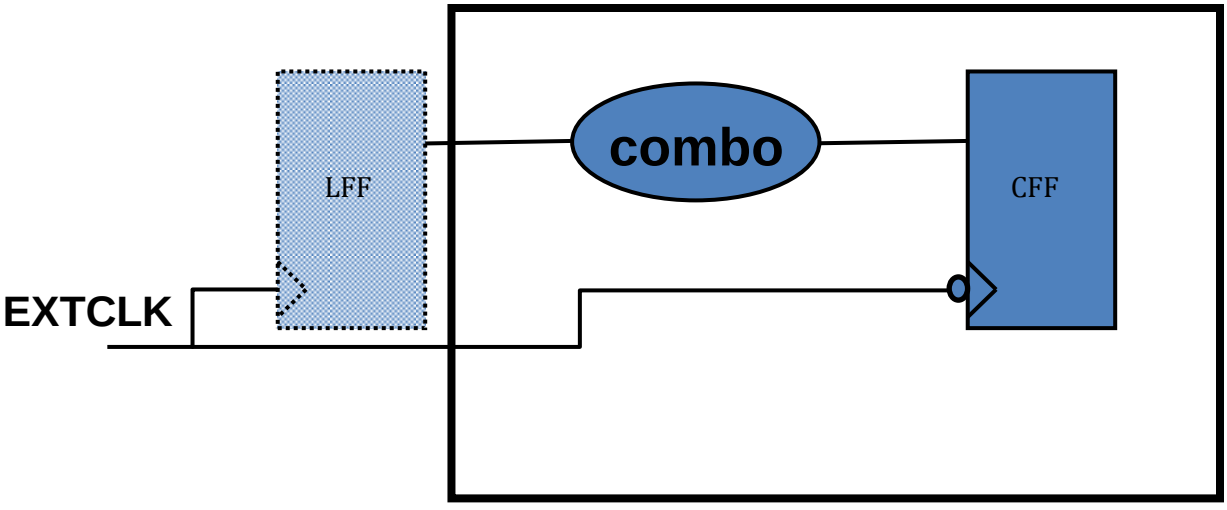

Figure 17: Simplified Schematic of Circuit

Applying fundamental #2 lets us understand the path, but attempting to apply the textbook start/end points won't work. We can't stay "`-from [get_pin LFF/CK]`" because that is a virtual pin and doesn't exist inside the design. Likewise, we can't say "`-to [get_pin CFF/CK]`" because a clock pin is not a valid endpoint. To solve this, we need to also apply fundamental #3 and leverage the expanded list of allowable start/end points – in this case, using a clock as our start and end point, and using rise_from/fall_from/rise_to/fall_to to select the appropriate edges.

```
report_timing –rise_from [get_clock EXTCLK] –fall_to \
   [get_clockEXTCLK] –through [get_pin CFF_1/D] –cap \
   -pathfull_clock_expanded-delay_type max
report_timing –fall_from [get_clock EXTCLK] –rise_to \
   [get_clockEXTCLK] –through [get_pin CFF_1/D] –cap \
   -pathfull_clock_expanded –delay_type min

report_timing –rise_from [get_clock EXTCLK] –fall_to \
   [get_clockEXTCLK] –through [get_pin CFF_2/D] –cap \
   -pathfull_clock_expanded –delay_type min
report_timing –fall_from [get_clock EXTCLK] –rise_to \
   [get_clockEXTCLK] –through [get_pin CFF_2/D] –cap \
   -pathfull_clock_expanded –delay_type max
```

4. Conclusion

Many users think only users of a dedicated static timing analysis tool like PrimeTime need to worry about static timing analysis. In truth, because most of our SOC implementation tools rely on static timing, every person involved in SOC design from library development, chip architecture, and through place and route should have an understanding of how STA works. Simplifying STA to its fundamentals makes this easier and more effective.

Too often users focus on the tool and its features to find solutions to problems, get frustrated when they are unable to get desired results from a tool, or have to wait for a vendor to confirm that a tool has a problem. In almost every case the solution can be found faster and with less effort by applying the three fundamentals of static timing analysis.

5. References

[i] "How Are Clock Gating Checks Inferred?" https://solvnet.synopsys.com/retrieve/015769.html

[ii] "Disable Auto Inferred Clock Gate Check" https://solvnet.synopsys.com/retrieve/028210.html