“No Man's Land”
Constraining async clock domain crossings

Paul Zimmer
Zimmer Design Services

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

ABSTRACT

Fifos using gray-coded pointers are a common technique for passing data between asynchronous clock domains. However, this technique has a hidden assumption that the skew between the bits is minimal relative to the clock periods involved. This assumption can be violated by P&R tools, and common STA techniques will not flag the problem, since they treat asynchronous domain crossings as unconstrained. This paper discusses the problem and proposes some techniques for constraining these paths.

2017.02.28: There may now be a better way to do all of this, now that Synopsys has given us "set_max_delay –ignore_clock_latency". I've experimented with it, and it seems to work. There are still details to be managed in order to fully automate the approach, but be aware that it exists. Be sure to use –allow_paths on set_clock groups.

Table of contents

Table of figures

1 Introduction - The Hidden Assumption in Synchronizing Fifos

Designers have long used gray code based fifos to transfer data safely between asynchronous clock domains. Such crossings are generally treated as "up to the designer" and not timed in static timing analysis. And yet there can be timing related issues that upset a fundamental assumption in these fifos and could cause them to fail. This paper proposes a method to check for this by constraining these paths.

The standard synchronizing fifo uses gray code counters as safe pointers for transfer across the asynchronous boundary. This works because the receiving side only sees one bit change at a time. So, it will either see the old value or the new value - both of which are valid.

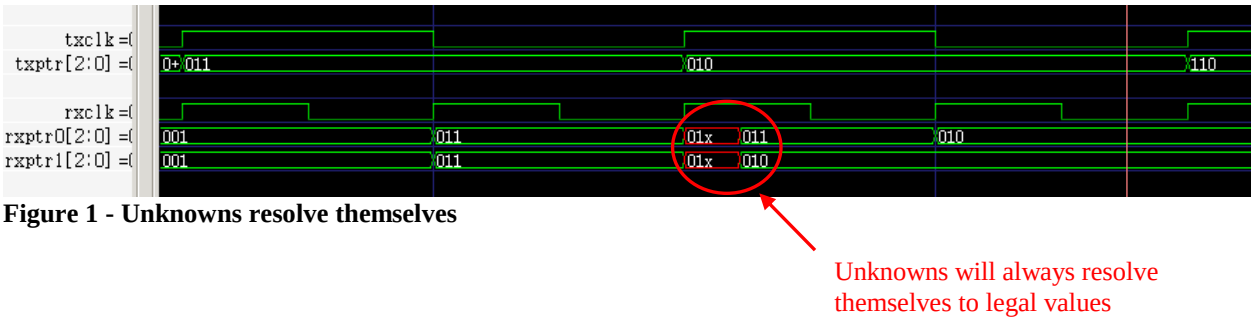

Figure 1 - Unknowns resolve themselves

In the example above, the transmitted pointer is going from 011 to 010 to 110. But the 011->010 transition occurs too close to the rxclk. The 2 possibilities are shown by rxptr0 and rxptr1. In both cases the value of bit 0 goes temporarily "x". In the case of rxptr0, it eventually resolves itself to the old value of 011. In the case of rxptr1, it eventually takes the new value of 010. If these 2 pointers are clocked again by rxclk before being used (a typical double synchronizer), then the receiving logic will still see 011->010. The only thing that varies is *which* rxclk edge will see the transition.

Contrast this to the case where the pointer had not been a gray code. Had the transition been 011->000, for example, both low order bits could experience sampling uncertainty and there would be 4 possible outcomes: 011, 010, 001, and 000. Two of these (010 and 001) might not be valid fifo entries.

1.1 When is a gray code no longer a gray code

But there's a hidden assumption here. This assumes that the skew between the bits is less than one tx clock period. If it is greater than one tx clock period, then multiple bits can change at the "same time" as seen in the rx clock domain and bad pointer values can occur.

Consider the following scenario. The tx ptr is going through the sequence 110->111->101->100, changing on every tx clock. txptr[0] is the low order bit. txptrD0[0] is this same bit delayed by just over one tx clock. The resulting pointer txptrD0 goes through the sequence 110->100->101...

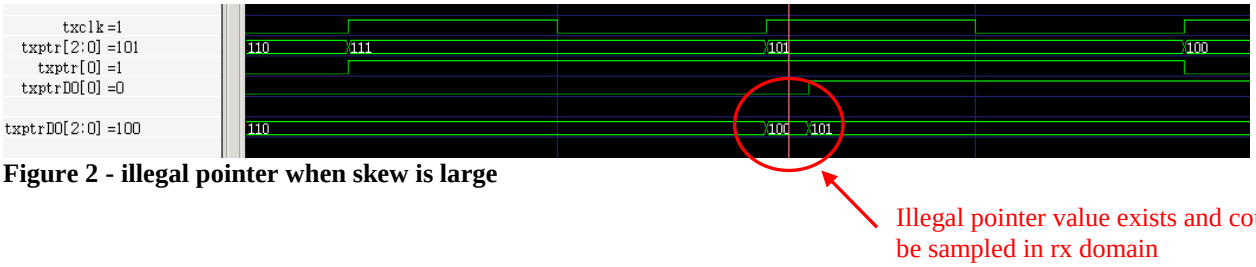

Figure 2 - illegal pointer when skew is large

The intermediate value 100 is not (at that time) a valid fifo location, and if the rx clock edge occurs during this period, the fifo may malfunction.

Since I have assumed zero delay on the other bits, the length of that "100" value is just exactly how much longer the delay on bit 0 is than one tx clock cycle. This will vary with PVT. If we assume that this trace represents some slow corner, one can easily imagine shrinking the duration of that "101" period down to zero at some PVT point. At that point, the entire gray code system will break down. Two bits will now be changing simultaneously, and bogus values can result.

This effect will also depend on how often the source pointer can change. If the source cannot change its pointer more often than every other tx clock, then our restriction would be that the pointer bits skew cannot be greater than 2 tx clock periods.

Note that, although I have used an absolute delay, the restriction is technically one of skew. If the delay on all the other bits were, say, 1 tx clock, then a delay of 1 tx clock plus a little on bit 0 would not cause a problem, but a delay of 2 tx clocks would. We have a problem when the skew across the pointer bits (at a given PVT point) exceeds one tx clock cycle.

1.2 Why is this a problem?

The problem with this is that standard STA techniques typically leave asynchronous boundary crossing unconstrained. All signals related to clocks that are synchronous to one another will be timed, but paths between asynchronous clocks cannot be timed in the normal way, since any arbitrary alignment of the clocks must be allowed and so all paths must fail.

This was not much of a problem back when layout tools were largely connection driven. But modern timing-driven layout tools are another matter. I have seen these tools do horrible things with unconstrained paths. And, since STA isn't looking at them either, we could get very large skews across the pointer bits which could result in malfunctioning synchronizing fifos.

2 Constraining these paths

Clearly, we need some way to constrain these paths so that we can catch cases where the skew between the pointer bits exceeds one tx clock period (or several tx clock periods, depending on the design).

2.1 Goals

We could, of course, use our knowledge of the detailed design to check these paths, using either report_timing or get_timing_paths. But this is generally a tedious operation, and requires that the STA engineer be told in detail about every such crossing. It also is an analysis-only approach, whereas it would be better if we could tell the layout tool not to do this in the first place.

It would be nice to have a general method for constraining these paths. The ideal solution would:

- Apply to all async clock crossing paths, and not require knowledge of specific flops or paths.
- Not require a timing update
- Not require any extracted data (like clock insertion delays) for creating the constraints
- Be independent of clock periods, other than that of the transmitting clock domain
- Be based on pure SDC and not be PrimeTime-specific (so that it can drive P&R)
- Should not interfere with correct timing of any paths within the normal STA, *including non-CDC paths that go to/from flops involved in the clock domain crossing*.
- Be correct from an SI perspective
- Be able to be fully automated

2.2 Constrain delay - not skew

Although the real requirement is less than one tx clock of skew between the gray code bits, I propose we simplify this to constrain for less than one tx clock of absolute delay from the sending clock to the receiving clock for all data bits. Designs that have to tolerate multiple clocks of delay are, I believe, rare. Also, PrimeTime has no way to constrain skew. True, you can use "set_data_check", but before you can apply it you have to find the fastest or slowest bit - meaning you've already done a timing update and calculated the skew. There's not a lot to be gained at that point by constraining it.

So, let's start with that simplification. We're going to constrain all async boundary crossings such that their paths cannot exceed one tx clock of delay. This will catch "scenic routes" as well as actual pointer failures so that we can examine these and repair or waive them.

2.3 Dealing with the clock insertion delays

Our goal of not having to extract clock insertion delays leads quickly to another conclusion - we're going to have do something special to eliminate or cancel the clock insertion delays. Every

constraining command in SDC uses a clock path for either launch or capture or both, and these delays are going to either have to be zero, or they're going to have to cancel.

2.3.1 Use leaf clocks

One idea I considered was to use Stuart Hecht's "leaf clocks" trick (see reference (3)). The idea here is to do a "create_generated_clock -comb myclk_leaf" on the clock pin of a flop in each clock network, then do "set_input_delay -clock myclk_leaf" on the internal nodes to be constrained. The effect is to put the same (approximately) insertion delay in both the launch and capture paths, thus making them cancel. This technique works great for timing i/o's, but for our use it has several drawbacks:

1. We have to do set_input_delay on all the receiving nodes, violating our goal of not having to either know or gather this information.
2. The use of set_input_delay on an internal pin, although it works in PrimeTime, is very non-standard and may not work the same way, or indeed at all, in other tools.
3. The set_input_delay will block unrelated paths through these pins, again violating one of our goals.
4. The cancellation is only approximate anyway

2.3.2 Use ideal clocks

The goal of avoiding knowledge of particular flops and paths makes solutions involving setting constraints on flops undesirable. We want to set some sort of constraint between clocks.

Try as I might, I could not find a way to constrain these paths using the normal STA clocks. Invariably the insertion delay would pop up on one side or the other (launch or capture) and distort the slack calculation. In the end, I concluded that it would be necessary to create ideal duplicates of the normal (real) STA clocks. Although a bit messy, it does have several advantages:

1. It keeps these "cdc" checks completely separate from the normal STA constraints
2. Unlike a virtual clock, these clocks "know" what flops are connected to them.

2.4 Finally - a use for set_max_delay!

I have long complained that the command "set_max_delay" is useless as currently defined (at least in PrimeTime. I have uses for it in synthesis). The problem is that the launch path always includes the clock propagation to the start point. This means that the target max delay value has to be adjusted to compensate for this by adding in this insertion delay. As the insertion delay varies with every layout, this requires either lots of messy data gathering (and another timing update), or manual intervention. In fact, Stuart Hecht's leaf clock trick mentioned above is a workaround for this exact problem.

To my great surprise, I found that set_max_delay is exactly the right command for the cdc constraint, but only if its usual bad habit of including the launch insertion delay can be tamed by forcing it to deal with only the ideal clock.

2.5 The basic idea

The basic idea is this:

1. Create a set of duplicate "_cdc"clocks in parallel to the normal STA clocks, but make them ideal.
2. Use set_max_delay from each of these clocks with a value equal to that clock's period (to get one tx clock of delay budget).
3. Control the paths between and among these clocks (and the STA clocks) to expose only the clock crossings we're trying to check.

2.6 Clock skew

One drawback to this technique is that it cannot account for clock skew. This hopefully should be much smaller than the delays we're constraining, but this technique cannot account for it. The only way to account for it is to adjust the max delay value. Since we want to avoid having to time the clock trees first, this would have to be done using a budget. We'll sweep this into our "fudge factor".

3 Examples

3.1 A basic circuit with two clocks

Here's a simple test circuit with 2 clock ports and a clock domain crossing (CDC):

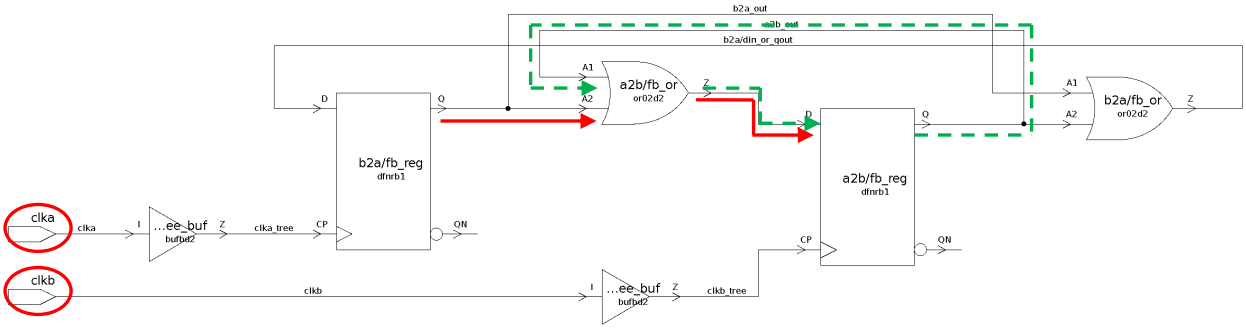

Figure 3 - Basic circuit with 2 clocks

Note that it has both a CDC path (in solid red) and a non-CDC path (in dashed green) so that we can verify that the non-CDC path is still timed correctly.

First, we'll set up basic STA. Create clka and clkb, choosing periods that don't line up:

```
&cmd create_clock -name clka -period 10.0 [get_ports clka]
&cmd create_clock -name clkb -period 3.3 [get_ports clkb]
```

If you don't recognize "&cmd", see Reference (1). It echoes the command as it executes.

Now put them in async clock groups:

```
&cmd set_clock_groups -async -group {clka} -group {clkb}
```

If I report timing through the dashed green path, I get a normal clkb timing check, as expected:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A1[1]

  Startpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Path Group: clkb
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clkb (rise edge)                   0.00       0.00
  clock network delay (propagated)         4.80       4.80
  a2b/fb_reg/CP (dfnrb1)                   0.00       4.80 r
  a2b/fb_reg/Q (dfnrb1)                    0.34       5.14 f
  a2b/fb_or/A1 (or02d2) <-                 0.00       5.14 f
  a2b/fb_or/Z (or02d2)                     0.14       5.28 f
  a2b/fb_reg/D (dfnrb1)                    0.00       5.28 f
  data arrival time                                   5.28

  clock clkb (rise edge)                   3.30       3.30
  clock network delay (propagated)         4.80       8.10
  a2b/fb_reg/CP (dfnrb1)                              8.10 r
  library setup time                      -0.08       8.02
  data required time                                  8.02
  -----------------------------------------------------------
  data required time                                  8.02
  data arrival time                                  -5.28
  -----------------------------------------------------------
  slack (MET)                                         2.75
```

If I report the red path, I get unconstrained, as expected:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A2
...
No constrained paths.
```

That was the path from clka to clkb and the clkb internal path (in dashed green). It holds true for the clka internal path and the clkb to clka path as well:

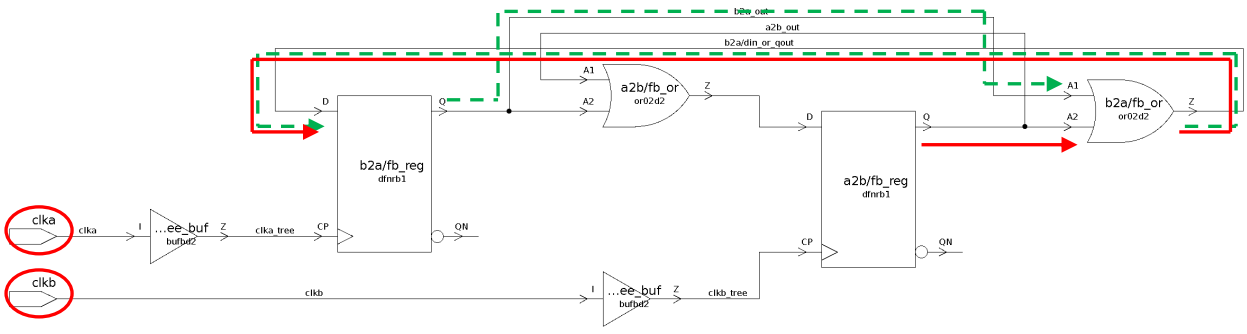

Figure 4 - Basic circuit with 2 clocks: alternative paths

[1] Why the complicated report_timing command? The -max, -nworst are to make sure I don't miss any paths, the -rise_through is so that I don't see 2 paths where I only wanted one, and -slack_less 1000 is because they change PT recently so that it reports automatically infers -slack_less 0 in most cases.

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2a/fb_or/A1
...
  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka)
  Endpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka)
  Path Group: clka
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clka (rise edge)                   0.00       0.00
  clock network delay (propagated)         1.40       1.40
  b2a/fb_reg/CP (dfnrb1)                   0.00       1.40 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       1.74 f
  b2a/fb_or/A1 (or02d2) <-                 0.00       1.74 f
  b2a/fb_or/Z (or02d2)                     0.14       1.88 f
  b2a/fb_reg/D (dfnrb1)                    0.00       1.88 f
  data arrival time                                   1.88

  clock clka (rise edge)                  10.00      10.00
  clock network delay (propagated)         1.40      11.40
  b2a/fb_reg/CP (dfnrb1)                             11.40 r
  library setup time                      -0.08      11.32
  data required time                                 11.32
  -----------------------------------------------------------
  data required time                                 11.32
  data arrival time                                  -1.88
  -----------------------------------------------------------
  slack (MET)                                         9.45

pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2a/fb_or/A2
...

No constrained paths.
```

To time the CDC paths, first we create _cdc versions of clka and clkb in parallel to the original clocks (using -add):

```
# Create the cdc clocks
&cmd create_clock -name clka_cdc -period [get_attribute [get_clocks clka]
period] [get_ports clka] -add
&cmd create_clock -name clkb_cdc -period [get_attribute [get_clocks clkb]
period] [get_ports clkb] -add
```

Note that I got the period of the original clock using get_attribute. If this is being hand-coded, you can just use the period.

Make sure they aren't propagated:

```
# Make sure cdc clocks aren't propagated
&cmd remove_propagated_clock [get_clocks *_cdc]
```

We don't really want the internal paths on the cdc clocks to get timed, because they lack correct clock insertion delay, which could result in bogus violations. Also we don't want to have to worry about false and multicycle paths internal to clka and clkb. If these are point-to-point exceptions, it won't matter. But if they use the clocks (like set_false_path -through <something> -to [get_clocks ...]), then we'd have to duplicate them for the cdc clocks.

These paths are being timed by the original clocks anyway, so just disable all internal timing on the cdc clocks:

```
# No internal paths on cdc clocks
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_false_path -from [get_clock $cdcclk] -to [get_clock $cdcclk]
}
```

We don't want our new cdc clocks to create noise in the original clock timing analysis, so make these clocks physically exclusive from all the other clocks:

```
# Make cdc clocks physically exclusive from all other clocks
&cmd set_clock_groups -physically_exclusive \
  -group [remove_from_collection [get_clocks *] [get_clocks *_cdc]] \
  -group [get_clocks *_cdc]
```

I'll address how to handle physically exclusive groups within the STA clocks later. This simple design doesn't have any.

Finally, we apply our set_max_delay:

```
# Use set_max_delay to apply a constraint of one tx clock to each cdc clock
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk
}
```

The value used is the period of the transmit clock. Since the constraint is "set_max_delay -from", this is the from clock.

Note that the "`[get_attribute $cdcclk period]`" might kick off a mini-timing update, so it would be better to keep your own copy of the clock periods in an array and use this array instead.

Better yet, include a fudge factor:

```
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [expr [get_attribute $cdcclk period] - $fudge] -from
$cdcclk
}
```

As you shall see, we're going to need it.

Now, let's report timing on the red path (figure 3) again:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A2
...
  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  b2a/fb_reg/CP (dfnrb1)                   0.00       0.00 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       0.34 r
  a2b/fb_or/A2 (or02d2) <-                 0.00       0.34 r
  a2b/fb_or/Z (or02d2)                     0.15       0.49 r
  a2b/fb_reg/D (dfnrb1)                    0.00       0.49 r
  data arrival time                                   0.49

  max_delay                               10.00      10.00
  library setup time                      -0.09       9.91
  data required time                                  9.91
  -----------------------------------------------------------
  data required time                                  9.91
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                        9.43
```

NO clock insertion delay!

Check is against 1 tx clock period

Now it's constrained! It is constrained so that the clock to q delay, plus the prop delay, plus the setup requirement of the flop, must be less than 10 ns - the clock period of clka (the transmit clock). And there are not messy clock insertion delays.

The clkb to clka cdc path is also correct, this time using the clock period of clkb (3.3ns):

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2a/fb_or/A2
...

  Startpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb_cdc)
  Endpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka_cdc)
  Path Group: clka_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  a2b/fb_reg/CP (dfnrb1)                   0.00       0.00 r
  a2b/fb_reg/Q (dfnrb1)                    0.34       0.34 r
  b2a/fb_or/A2 (or02d2) <-                 0.00       0.34 r
  b2a/fb_or/Z (or02d2)                     0.15       0.49 r
  b2a/fb_reg/D (dfnrb1)                    0.00       0.49 r
  data arrival time                                   0.49

  max_delay                                3.30       3.30
  library setup time                      -0.09       3.21
  data required time                                  3.21
  -----------------------------------------------------------
  data required time                                  3.21
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                        2.73
```

Yeah!

And the dashed green path is unchanged:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A1
...
  Startpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Path Group: clkb
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clkb (rise edge)                   0.00       0.00
  clock network delay (propagated)         4.80       4.80
  a2b/fb_reg/CP (dfnrb1)                   0.00       4.80 r
  a2b/fb_reg/Q (dfnrb1)                    0.34       5.14 f
  a2b/fb_or/A1 (or02d2) <-                 0.00       5.14 f
  a2b/fb_or/Z (or02d2)                     0.14       5.28 f
  a2b/fb_reg/D (dfnrb1)                    0.00       5.28 f
  data arrival time                                   5.28

  clock clkb (rise edge)                   3.30       3.30
  clock network delay (propagated)         4.80       8.10
  a2b/fb_reg/CP (dfnrb1)                              8.10 r
  library setup time                      -0.08       8.02
  data required time                                  8.02
  -----------------------------------------------------------
  data required time                                  8.02
  data arrival time                                  -5.28
  -----------------------------------------------------------
  slack (MET)                                         2.75
```

3.2 What about i/o paths?

Let's take a slight variation of this circuit, but add i/o:

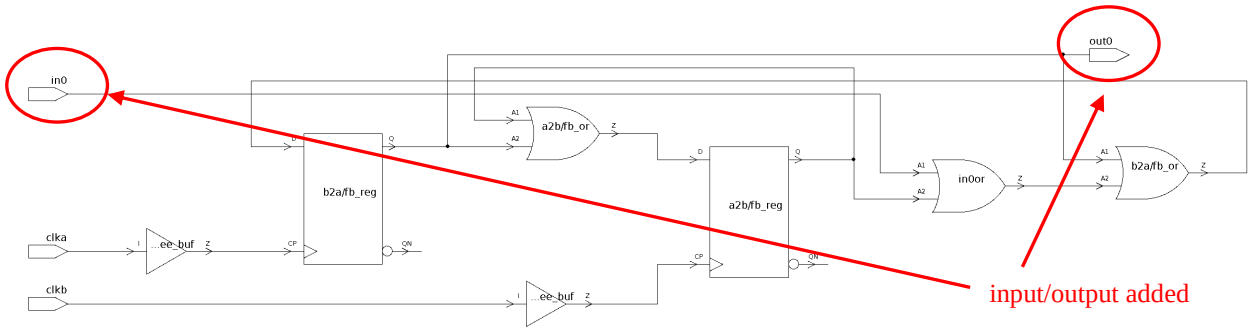

Figure 5 - Basic circuit with i/o

If we apply the constraints above and report timing to out0, we see that our set_max_delay has inadvertently constrained the port:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_to out0
...
  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: out0 (output port)
  Path Group: **default**
  Path Type: max

  Point                                              Incr       Path
  -----------------------------------------------------------------------
  b2a/fb_reg/CP (dfnrb1)                             0.00       0.00 r
  b2a/fb_reg/Q (dfnrb1)                              0.34       0.34 f
  out0 (out)                                         0.00       0.34 f
  data arrival time                                             0.34

  max_delay                                         10.00      10.00
  output external delay                              0.00      10.00
  data required time                                           10.00
  -----------------------------------------------------------------------
  data required time                                           10.00
  data arrival time                                            -0.34
  -----------------------------------------------------------------------
  slack (MET)                                                   9.66
```

Since the set_max_delay was apply as "-from $cdcclk", it constrains *all* paths from the cdc clock. But we only care about paths to flops, so it would be better to do this:

```
set all_regs_data_pins [all_registers -data_pins]
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk -to
$all_regs_data_pins
}
```

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_to out0
...
No constrained paths.
```

Of course, all_regs_data_pins could be a very large collection. This did not seem to create any problems on the medium-size chip I tried it on, but an alternative would be to create false paths from the _cdc clocks to all outputs:

```
&eval set_false_path -from [get_clocks *_cdc] -to [all_outputs]
```

This approach would probably be preferred if the sdc is going to be used in synthesis and then written out for P&R, since the all_regs_data_pins approach might result in a very large output sdc file.

There's no need to do anything about input paths, since our set_max_delay is *from* the _cdc clocks, and we haven't apply any input delays from the _cdc clocks.

3.3 The original circuit, but with another physically-exclusive clock

Now let's add a new wrinkle.

Suppose we were going to analyze another mode in parallel that involves a different clock frequency on clka (pointless on this circuit, I know, but it's just an example). Let's create a clkc that runs in parallel to clka:

```
# Make clkc on top of clk a
&cmd create_clock -period 5.5 -name clkc [get_ports clka]  -add
```

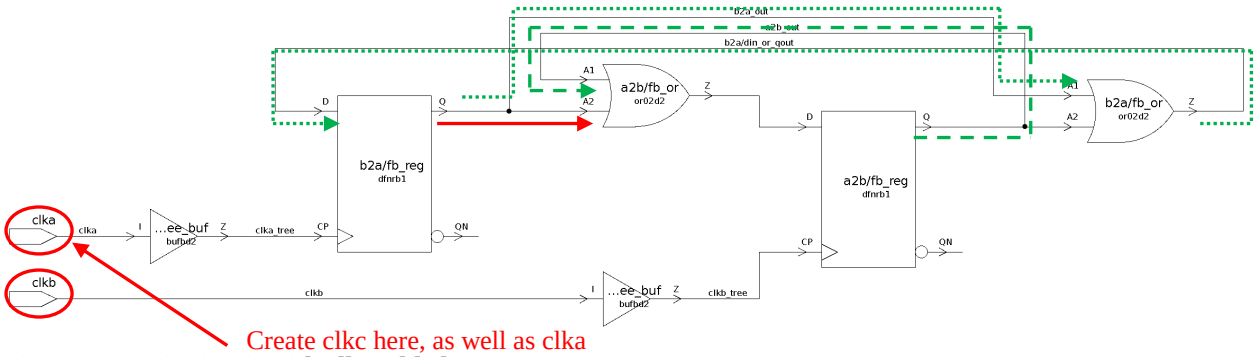

Figure 6 - Basic circuit with clkc added

If you're doing all of this manually, you could, of course, choose to create only clka_cdc or clkc_cdc based on which has the smaller period. I'm just using this as an example to illustrate something about clocks propagating in parallel.

When clkc is running, it is async to clkb just like clka:

```
&cmd set_clock_groups -async -group {clkb} -group {clkc}
```

But clka and clkc are never physically present at the same time, so they are *physically exclusive*:

```
&cmd set_clock_groups -phys -group {clka} -group {clkc}
```

If we time the dashed green path in figure 6 again, we still get:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A1
...
  Startpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Path Group: clkb
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clkb (rise edge)                   0.00       0.00
  clock network delay (propagated)         4.80       4.80
  a2b/fb_reg/CP (dfnrb1)                   0.00       4.80 r
  a2b/fb_reg/Q (dfnrb1)                    0.34       5.14 f
  a2b/fb_or/A1 (or02d2) <-                 0.00       5.14 f
  a2b/fb_or/Z (or02d2)                     0.14       5.28 f
  a2b/fb_reg/D (dfnrb1)                    0.00       5.28 f
  data arrival time                                   5.28

  clock clkb (rise edge)                   3.30       3.30
  clock network delay (propagated)         4.80       8.10
  a2b/fb_reg/CP (dfnrb1)                              8.10 r
  library setup time                      -0.08       8.02
  data required time                                  8.02
  -----------------------------------------------------------
  data required time                                  8.02
  data arrival time                                  -5.28
  -----------------------------------------------------------
  slack (MET)                                         2.75
```

If we time the *dotted* green path (internal feedback paths along the clka route), we now get one path each for clka and clkc:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2a/fb_or/A1
...

  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clkc)
  Endpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clkc)
  Path Group: clkc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clkc (rise edge)                   0.00       0.00
  clock network delay (propagated)         1.40       1.40
  b2a/fb_reg/CP (dfnrb1)                   0.00       1.40 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       1.74 r
  b2a/fb_or/A1 (or02d2) <-                 0.00       1.74 r
  b2a/fb_or/Z (or02d2)                     0.13       1.86 r
  b2a/fb_reg/D (dfnrb1)                    0.00       1.86 r
  data arrival time                                   1.86

  clock clkc (rise edge)                   5.50       5.50
  clock network delay (propagated)         1.40       6.90
  b2a/fb_reg/CP (dfnrb1)                              6.90 r
  library setup time                      -0.09       6.81
  data required time                                  6.81
  -----------------------------------------------------------
  data required time                                  6.81
  data arrival time                                  -1.86
  -----------------------------------------------------------
  slack (MET)                                         4.95
```

```
  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka)
  Endpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka)
  Path Group: clka
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clka (rise edge)                   0.00       0.00
  clock network delay (propagated)         1.40       1.40
  b2a/fb_reg/CP (dfnrb1)                   0.00       1.40 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       1.74 r
  b2a/fb_or/A1 (or02d2) <-                 0.00       1.74 r
  b2a/fb_or/Z (or02d2)                     0.13       1.86 r
  b2a/fb_reg/D (dfnrb1)                    0.00       1.86 r
  data arrival time                                   1.86

  clock clka (rise edge)                  10.00      10.00
  clock network delay (propagated)         1.40      11.40
  b2a/fb_reg/CP (dfnrb1)                             11.40 r
  library setup time                      -0.09      11.31
  data required time                                 11.31
  -----------------------------------------------------------
  data required time                                 11.31
  data arrival time                                  -1.86
  -----------------------------------------------------------
  slack (MET)                                         9.45
```

The red paths still report unconstrained:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A2
...

No constrained paths.

pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2a/fb_or/A2
...

No constrained paths.
```

To constrain these paths, we create parallel _cdc clocks as before:

```
# Create the cdc clocks
&cmd create_clock -name clka_cdc -period [get_attribute [get_clocks clka]
period] [get_ports clka] -add
&cmd create_clock -name clkb_cdc -period [get_attribute [get_clocks clkb]
period] [get_ports clkb] -add
&cmd create_clock -name clkc_cdc -period [get_attribute [get_clocks clkc]
period] [get_ports clka] -add
```

Now we make them unpropagated, disable their internal paths, set them physically exclusive from the main clocks, and apply the set_max_delay using the same code as before:

```
# Make sure cdc clocks aren't propagated
&cmd remove_propagated_clock [get_clocks *_cdc]

# No internal paths on cdc clocks
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_false_path -from [get_clock $cdcclk] -to [get_clock $cdcclk]
}

# Make cdc clocks physically exclusive from all other clocks
&cmd set_clock_groups -physically_exclusive \
  -group [remove_from_collection [get_clocks *] [get_clocks *_cdc]] \
  -group [get_clocks *_cdc]

# Use set_max_delay to apply a constraint of one tx clock to each cdc clock
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk
}
```

That will leave clka_cdc to time against clkb_cdc as before. And now clkc_cdc will time against clkb_cdc as well, representing the new async crossing.

But what about clka_cdc and clkc_cdc? Should they time against each other?

Of course not. clka and clkc are physically exclusive, so clka_cdc and clkc_cdc must be physically exclusive as well:

```
&cmd set_clock_groups -physically_exclusive -group {clka_cdc} -group
{clkc_cdc}
```

This raises an important point that will be useful when we try to automate all of this:

If the original STA clocks are physically or logically exclusive, then the _cdc versions will be physically or logically exclusive as well.

Note that this is *not* true of async clocks! The whole point of this is to time those async boundaries, so those paths *must* be left enabled.

Now let's look at the red path (in figure 6):

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A2
...
  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clkc_cdc)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  b2a/fb_reg/CP (dfnrb1)                   0.00       0.00 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       0.34 r
  a2b/fb_or/A2 (or02d2) <-                 0.00       0.34 r
  a2b/fb_or/Z (or02d2)                     0.15       0.49 r
  a2b/fb_reg/D (dfnrb1)                    0.00       0.49 r
  data arrival time                                   0.49

  max_delay                                5.50       5.50
  library setup time                      -0.09       5.41
  data required time                                  5.41
  -----------------------------------------------------------
  data required time                                  5.41
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                         4.93


  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  b2a/fb_reg/CP (dfnrb1)                   0.00       0.00 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       0.34 r
  a2b/fb_or/A2 (or02d2) <-                 0.00       0.34 r
  a2b/fb_or/Z (or02d2)                     0.15       0.49 r
  a2b/fb_reg/D (dfnrb1)                    0.00       0.49 r
  data arrival time                                   0.49

  max_delay                               10.00      10.00
  library setup time                      -0.09       9.91
  data required time                                  9.91
  -----------------------------------------------------------
  data required time                                  9.91
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                         9.43
```

Now we get one path constrained using clkc_cdc (with a constraint of 5.5ns) and one using clka_cdc (with a constraint of 10 ns). Just what we wanted.

3.4 More than one clock in an async clock group - adding a divided clock

Now let's see what happens when there is more than one clock in each group. As a simple example, consider this circuit, where I have added a divide-by 2 clock on clka:

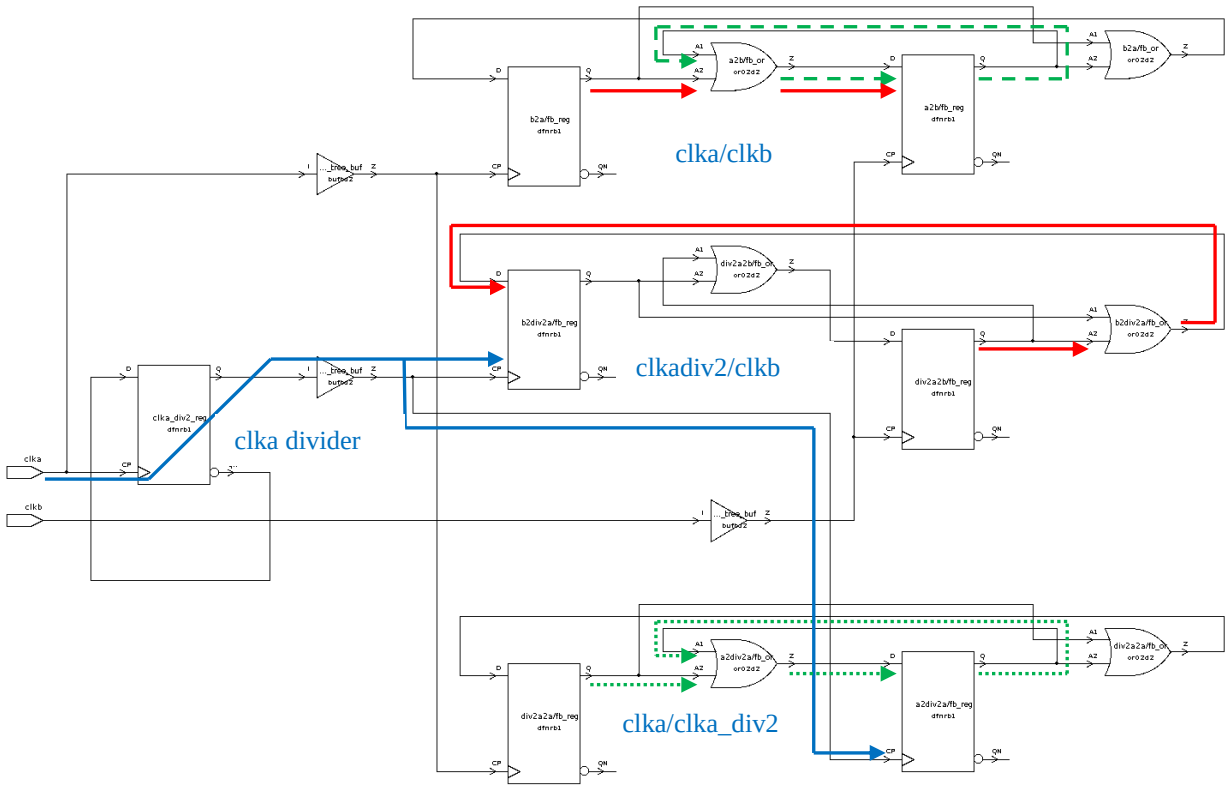

Figure 7 - Generated clock circuit

In addition to the original clka/clkb crossing, I have added a clka_div2/clkb crossing, *and* a clka/clka_div2 crossing (dotted green).

For normal STA, we create the clocks as before, but adding the divide-by 2 clock on clka:

```
# Create the clocks
create_clock -name clka -period 10.0 [get_ports clka]
create_clock -name clkb -period 3.3 [get_ports clkb]
create_generated_clock -name clka_div2 -divide_by 2 -master clka -add -source
[get_attribute [get_clocks clka] sources] [get_pins clka_div2_reg/Q]
```

We know that clka and clkb are asynchronous, so clka_div2 and clkb must be asynchronous as well. We'll use set_clock_groups with a wildcard:

```
&cmd set_clock_groups -async -group {clka*} -group {clkb}
```

Following the pattern we used earlier, we create cdc versions of these clocks:

```
# Create the cdc clocks
&cmd create_clock -name clka_cdc -period [get_attribute [get_clocks clka]
period] [get_ports clka] -add
&cmd create_clock -name clkb_cdc -period [get_attribute [get_clocks clkb]
period] [get_ports clkb] -add
create_generated_clock -name clka_div2_cdc -divide_by 2 -master clka_cdc -add
-source [get_attribute [get_clocks clka_cdc] sources] [get_pins
clka_div2_reg/Q]
```

And set them propagated, etc as before:

```
# Make sure cdc clocks aren't propagated
&cmd remove_propagated_clock [get_clocks *_cdc]

# No internal paths on cdc clocks
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_false_path -from [get_clock $cdcclk] -to [get_clock $cdcclk]
}

# Make cdc clocks physically exclusive from all other clocks
&cmd set_clock_groups -physically_exclusive \
  -group [remove_from_collection [get_clocks *] [get_clocks *_cdc]] \
  -group [get_clocks *_cdc]

# Use set_max_delay to apply a constraint of one tx clock to each cdc clock
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk
}

&eval set_false_path -from [get_clocks *_cdc] -to [all_outputs]
```

If we time the original clka/clkb paths, nothing has changed:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A1
...
  Startpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb)
  Path Group: clkb
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clkb (rise edge)                   0.00       0.00
  clock network delay (propagated)         4.80       4.80
  a2b/fb_reg/CP (dfnrb1)                   0.00       4.80 r
  a2b/fb_reg/Q (dfnrb1)                    0.35       5.15 f
  a2b/fb_or/A1 (or02d2) <-                 0.00       5.15 f
  a2b/fb_or/Z (or02d2)                     0.14       5.28 f
  a2b/fb_reg/D (dfnrb1)                    0.00       5.28 f
  data arrival time                                   5.28

  clock clkb (rise edge)                   3.30       3.30
  clock network delay (propagated)         4.80       8.10
  a2b/fb_reg/CP (dfnrb1)                              8.10 r
  library setup time                      -0.07       8.03
  data required time                                  8.03
  -----------------------------------------------------------
  data required time                                  8.03
  data arrival time                                  -5.28
  -----------------------------------------------------------
  slack (MET)                                         2.75


pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2b/fb_or/A2
...


  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  b2a/fb_reg/CP (dfnrb1)                   0.00       0.00 r
  b2a/fb_reg/Q (dfnrb1)                    0.34       0.34 r
  a2b/fb_or/A2 (or02d2) <-                 0.00       0.34 r
  a2b/fb_or/Z (or02d2)                     0.15       0.49 r
  a2b/fb_reg/D (dfnrb1)                    0.00       0.49 r
  data arrival time                                   0.49

  max_delay                               10.00      10.00
  library setup time                      -0.08       9.92
  data required time                                  9.92
  -----------------------------------------------------------
  data required time                                  9.92
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                         9.43
```

Now look at the new crossing, clka_div2/clkb.

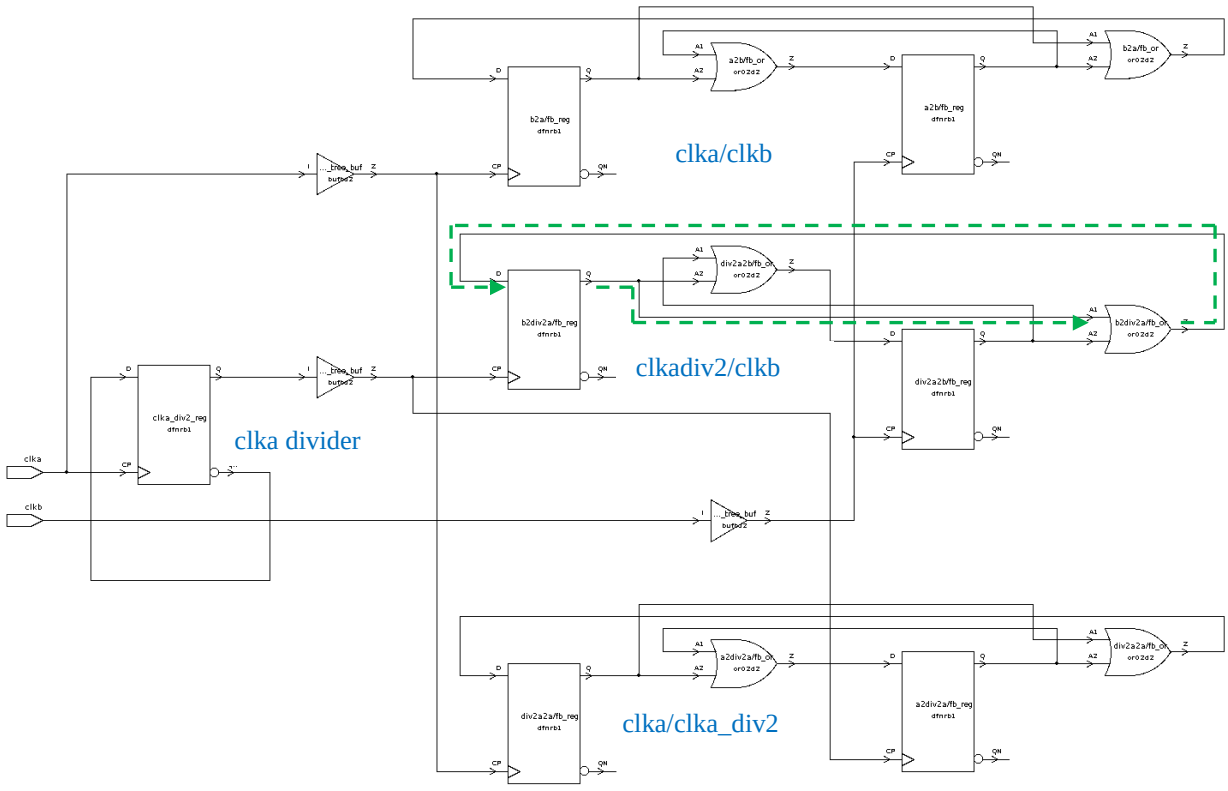

Figure 8 - Generated clock circuit - clkadiv2 internal paths

First, the dashed green path (clka_div2 internal path):

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2div2a/fb_or/A1
...
  Startpoint: b2div2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_div2)
  Endpoint: b2div2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_div2)
  Path Group: clka_div2
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clka_div2 (rise edge)              0.00       0.00
  clock network delay (propagated)         5.11       5.11
  b2div2a/fb_reg/CP (dfnrb1)               0.00       5.11 r
  b2div2a/fb_reg/Q (dfnrb1)                0.35       5.45 f
  b2div2a/fb_or/A1 (or02d2) <-             0.00       5.45 f
  b2div2a/fb_or/Z (or02d2)                 0.14       5.59 f
  b2div2a/fb_reg/D (dfnrb1)                0.00       5.59 f
  data arrival time                                   5.59

  clock clka_div2 (rise edge)             20.00      20.00
  clock network delay (propagated)         5.11      25.11
  b2div2a/fb_reg/CP (dfnrb1)                         25.11 r
  library setup time                      -0.07      25.04
  data required time                                 25.04
  -----------------------------------------------------------
  data required time                                 25.04
  data arrival time                                  -5.59
  -----------------------------------------------------------
  slack (MET)                                        19.45
```

This is a normal timing check, complete with clock trees. The period is 2x the period of clka - or 20 ns.

On the crossing paths between clkadiv2 and clkb, we get the expected checks for path less than tx clock period:

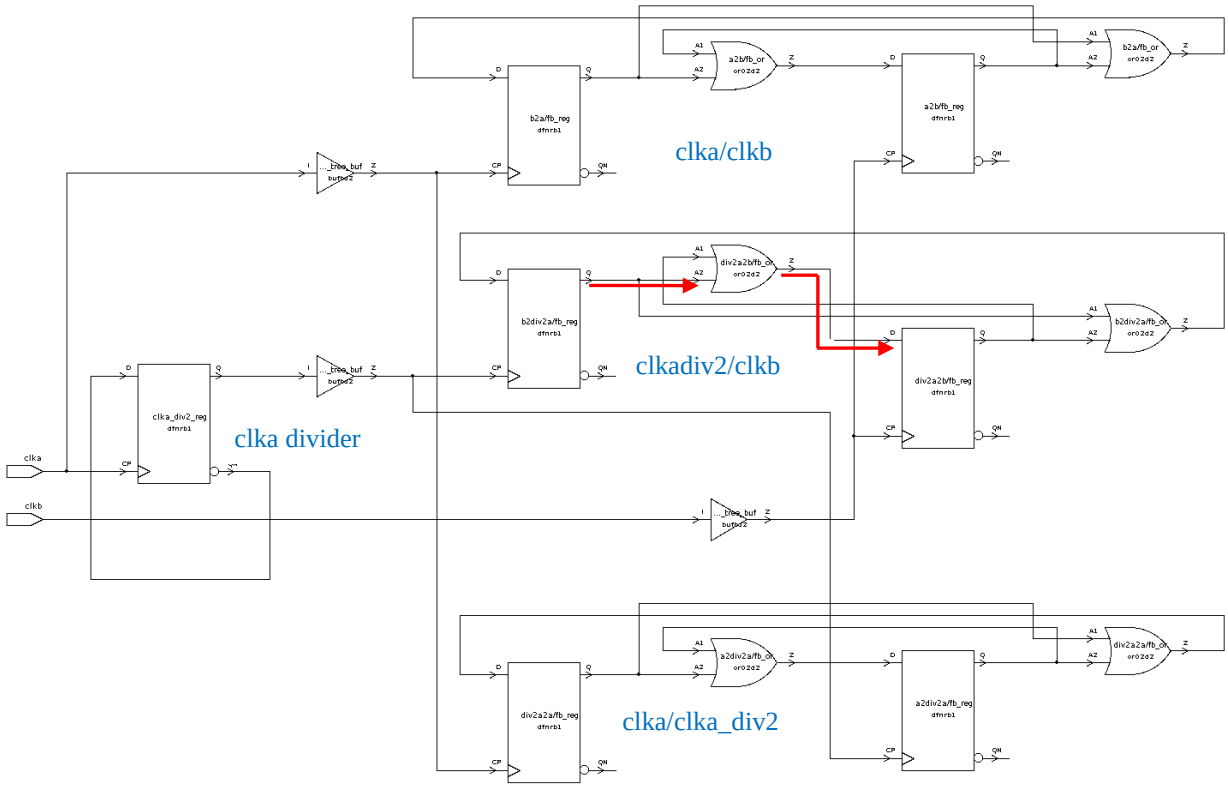

Figure 9 - Generated clock circuit - clkadiv2/clkb crossing paths

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through b2div2a/fb_or/A2
...

  Startpoint: div2a2b/fb_reg
              (rising edge-triggered flip-flop clocked by clkb_cdc)
  Endpoint: b2div2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_div2_cdc)
  Path Group: clka_div2_cdc
  Path Type: max

  Point                                     Incr       Path
  -----------------------------------------------------------
  div2a2b/fb_reg/CP (dfnrb1)                0.00       0.00 r
  div2a2b/fb_reg/Q (dfnrb1)                 0.34       0.34 r
  b2div2a/fb_or/A2 (or02d2) <-              0.00       0.34 r
  b2div2a/fb_or/Z (or02d2)                  0.15       0.49 r
  b2div2a/fb_reg/D (dfnrb1)                 0.00       0.49 r
  data arrival time                                    0.49

  max_delay                                 3.30       3.30
  library setup time                       -0.08       3.22
  data required time                                   3.22
  -----------------------------------------------------------
  data required time                                   3.22
  data arrival time                                   -0.49
  -----------------------------------------------------------
  slack (MET)                                          2.73

pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through div2a2b/fb_or/A2
...

  Startpoint: b2div2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_div2_cdc)
  Endpoint: div2a2b/fb_reg
              (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                     Incr       Path
  -----------------------------------------------------------
  b2div2a/fb_reg/CP (dfnrb1)                0.00       0.00 r
  b2div2a/fb_reg/Q (dfnrb1)                 0.34       0.34 r
  div2a2b/fb_or/A2 (or02d2) <-              0.00       0.34 r
  div2a2b/fb_or/Z (or02d2)                  0.15       0.49 r
  div2a2b/fb_reg/D (dfnrb1)                 0.00       0.49 r
  data arrival time                                    0.49

  max_delay                                20.00      20.00
  library setup time                       -0.08      19.92
  data required time                                  19.92
  -----------------------------------------------------------
  data required time                                  19.92
  data arrival time                                   -0.49
  -----------------------------------------------------------
  slack (MET)                                         19.43
```

But now look at the clka/clka_div2 path:

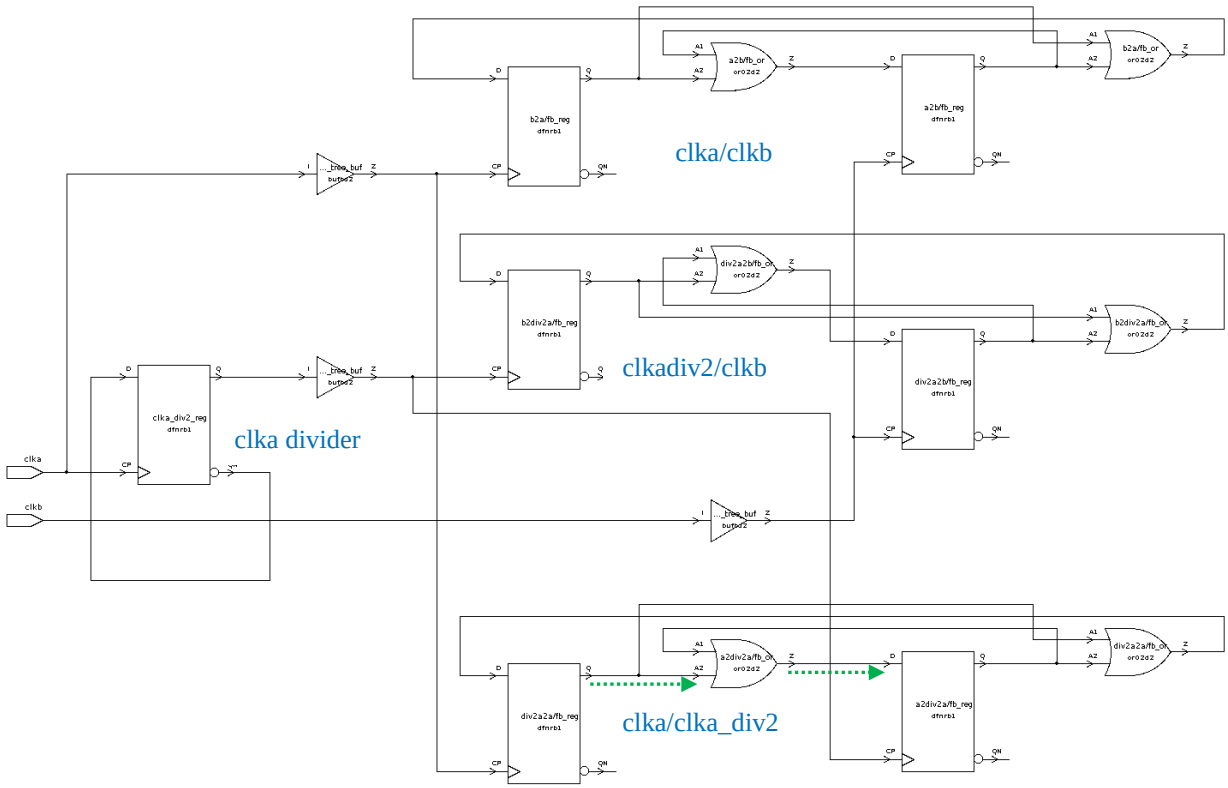

Figure 10 - Generated clock circuit - clka/clka_div2 path

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2div2a/fb_or/A2
...

  Startpoint: div2a2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2div2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_div2_cdc)
  Path Group: clka_div2_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  div2a2a/fb_reg/CP (dfnrb1)               0.00       0.00 r
  div2a2a/fb_reg/Q (dfnrb1)                0.34       0.34 r
  a2div2a/fb_or/A2 (or02d2) <-             0.00       0.34 r
  a2div2a/fb_or/Z (or02d2)                 0.15       0.49 r
  a2div2a/fb_reg/D (dfnrb1)                0.00       0.49 r
  data arrival time                                   0.49

  max_delay                               10.00      10.00
  library setup time                      -0.08       9.92
  data required time                                  9.92
  -----------------------------------------------------------
  data required time                                  9.92
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                        9.43
```

cdc checks are only for *cross-clock* paths!

```
Startpoint: div2a2a/fb_reg
            (rising edge-triggered flip-flop clocked by clka)
Endpoint: a2div2a/fb_reg
            (rising edge-triggered flip-flop clocked by clka_div2)
Path Group: clka_div2
Path Type: max

Point                                     Incr       Path
-----------------------------------------------------------
clock clka (rise edge)                   10.00      10.00
clock network delay (propagated)          1.40      11.40
div2a2a/fb_reg/CP (dfnrb1)                0.00      11.40 r
div2a2a/fb_reg/Q (dfnrb1)                 0.34      11.74 r
a2div2a/fb_or/A2 (or02d2) <-              0.00      11.74 r
a2div2a/fb_or/Z (or02d2)                  0.15      11.89 r
a2div2a/fb_reg/D (dfnrb1)                 0.00      11.89 r
data arrival time                                   11.89

clock clka_div2 (rise edge)              20.00      20.00
clock network delay (propagated)          5.11      25.11
a2div2a/fb_reg/CP (dfnrb1)                          25.11 r
library setup time                       -0.08      25.03
data required time                                  25.03
-----------------------------------------------------------
data required time                                  25.03
data arrival time                                  -11.89
-----------------------------------------------------------
slack (MET)                                         13.13
```

We failed to disable the cdc path!

Remember, we only want the cdc clocks to time paths that pass *between* clock groups, never *within* them. We already disabled paths within each _cdc clock. Now we must extend this to internal paths within each _cdc clock's group. This raises another important point that will be useful when we try to automate this technique:

Each clock within each asynchronous clock group in the original STA clock groups must be set logically exclusive to all the other clocks in that group.

In this case, that is just clka and clka_div2:

```
&cmd set_clock_groups -logically_exclusive -group {clka_cdc} -group
{clka_div2_cdc}
```

We use -logically_exclusive instead of -asynchronous because we *probably* want PT to use SI windows that follow the clock waveforms, since these are actually synchronous clocks. The "*probably*" part will be explained later in the SI discussion.

Now, if we run the report, we get only the expected timing report:

```
pt_shell> report_timing -max 10 -nworst 10 -slack_less 1000 -rise_through a2div2a/fb_or/A2
****************************************
Report : timing
        -path_type full
        -delay_type max
        -max_paths 1
Design : two_clock_div2a
Version: D-2009.12-SP1
Date   : Mon Jul  9 14:58:20 2012
****************************************


  Startpoint: div2a2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka)
  Endpoint: a2div2a/fb_reg
              (rising edge-triggered flip-flop clocked by clka_div2)
  Path Group: clka_div2
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clka (rise edge)                  10.00      10.00
  clock network delay (propagated)         1.40      11.40
  div2a2a/fb_reg/CP (dfnrb1)               0.00      11.40 r
  div2a2a/fb_reg/Q (dfnrb1)                0.34      11.74 r
  a2div2a/fb_or/A2 (or02d2) <-             0.00      11.74 r
  a2div2a/fb_or/Z (or02d2)                 0.15      11.89 r
  a2div2a/fb_reg/D (dfnrb1)                0.00      11.89 r
  data arrival time                                  11.89

  clock clka_div2 (rise edge)             20.00      20.00
  clock network delay (propagated)         5.11      25.11
  a2div2a/fb_reg/CP (dfnrb1)                         25.11 r
  library setup time                      -0.08      25.03
  data required time                                 25.03
  -----------------------------------------------------------
  data required time                                 25.03
  data arrival time                                 -11.89
  -----------------------------------------------------------
  slack (MET)                                        13.13
```

Note that the issue here is quite general. If the original STA had a clock group like this:

```
set_clock_groups -async -group {clk1 clk2} -group {clk3 clk4 clk5}
```

We would need to generate cdc clock groups like this:

```
set_clock_groups -log -group {clk1_cdc} -group {clk2_cdc}
set_clock_groups -log -group {clk3_cdc} -group {clk4_cdc} -group {clk5_cdc}
```

4 But what if I DO want to target specific flops?

OK, so we now have a mechanism for constraining all paths between all async clocks to have one tx clock cycle of delay or less. This is, of course, overly restrictive. It might result in a lot of paths we don't care about, like config bits. One (conservative) way to deal with this is to look at the paths one-by-one and waive them. But suppose the design flow is well controlled and we know what paths involve the grey code counters. What if we only want to constrain specific paths?

Well, this is more complicated that it looks. Consider this simple example:

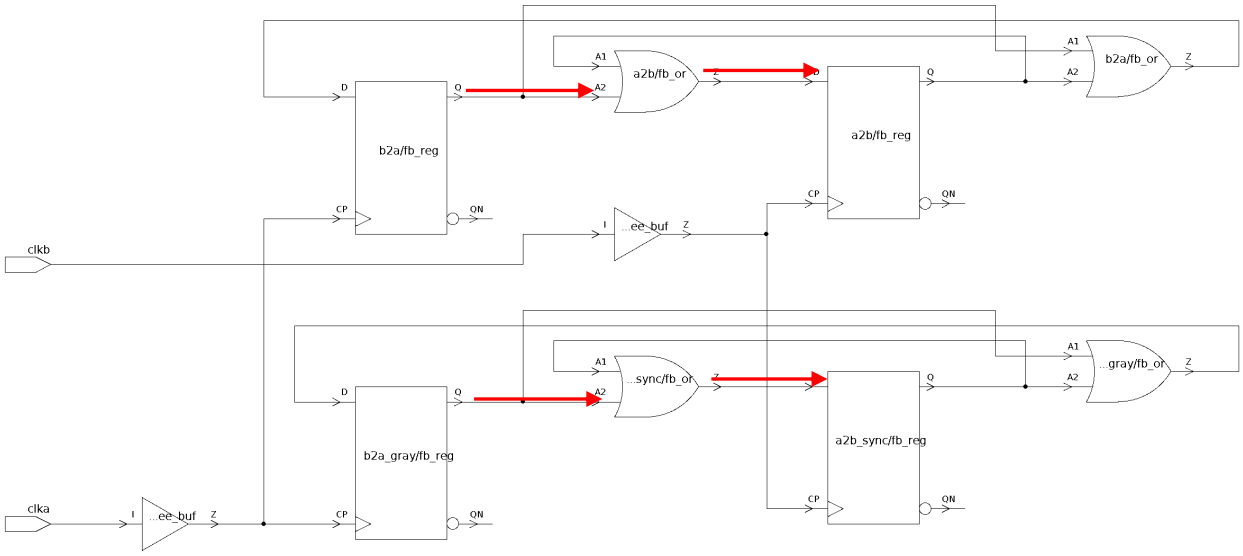

Figure 11 - Targeting specific flops

The top path (b2a/fb_reg to a2b/fb_reg) is a non-critical path and should remain unconstrained. The bottom path (b2a_gray/fb_reg to a2b_sync/fb_reg) is a gray-code path and should be constrained.

In order to get set_max_delay to work correctly (not have clock insertion delay), you need to use the _cdc clocks, and their cross-clock paths must be enabled as described above. When constraining all paths, we can use set_max_delay to override the normal check. But if we apply the set_max_delay only to the target path, the normal check comes back:

```
set gray_flops [filter_collection [all_registers] "full_name =~ *_gray/*"]
# This won't do - it leaves the normal flops with constrained paths.
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk -through
$gray_flops
}
```

```
pt_shell> report_timing -max 10 -nworst 1 -slack_less 1000 -rise_to [get_clocks clkb_cdc]
...
  Startpoint: b2a/fb_reg (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2b/fb_reg (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clka_cdc (rise edge)             320.00     320.00
  clock network delay (ideal)              0.00     320.00
  b2a/fb_reg/CP (dfnrb1)                   0.00     320.00 r
  b2a/fb_reg/Q (dfnrb1)                    0.34     320.34 r
  a2b/fb_or/Z (or02d2)                     0.15     320.49 r
  a2b/fb_reg/D (dfnrb1)                    0.00     320.49 r
  data arrival time                                 320.49

  clock clkb_cdc (rise edge)             320.10     320.10
  clock network delay (ideal)              0.00     320.10
  a2b/fb_reg/CP (dfnrb1)                           320.10 r
  library setup time                      -0.08     320.02
  data required time                                320.02
  -----------------------------------------------------------
  data required time                                320.02
  data arrival time                                -320.49
  -----------------------------------------------------------
  slack (VIOLATED)                                   -0.47


  Startpoint: b2a_gray/fb_reg
              (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2b_sync/fb_reg
              (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  b2a_gray/fb_reg/CP (dfnrb1)              0.00       0.00 r
  b2a_gray/fb_reg/Q (dfnrb1)               0.34       0.34 r
  a2b_sync/fb_or/Z (or02d2)                0.15       0.49 r
  a2b_sync/fb_reg/D (dfnrb1)               0.00       0.49 r
  data arrival time                                   0.49

  max_delay                               10.00      10.00
  library setup time                      -0.08       9.92
  data required time                                  9.92
  -----------------------------------------------------------
  data required time                                  9.92
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                         9.43
```


We got our constraint on the gray to sync path as expected, but without a set_max_delay to override the normal check, PT looks for the worst-case clock alignment of the async clocks and does a normal check.

So, we need to explicitly make these paths false:

```
# Must disable non_gray_flops
set gray_flops [filter_collection [all_registers] "full_name =~ *_gray/*"]
set non_gray_flops [remove_from_collection [all_registers] $gray_flops]
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk -through
$gray_flops
  &cmd set_false_path -from $cdcclk -through $non_gray_flops
}
```

Now the bad path is gone:

```
pt_shell> report_timing -max 10 -nworst 1 -slack_less 1000 -rise_to [get_clocks clkb_cdc]
...
  Startpoint: b2a_gray/fb_reg
              (rising edge-triggered flip-flop clocked by clka_cdc)
  Endpoint: a2b_sync/fb_reg
            (rising edge-triggered flip-flop clocked by clkb_cdc)
  Path Group: clkb_cdc
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  b2a_gray/fb_reg/CP (dfnrb1)              0.00       0.00 r
  b2a_gray/fb_reg/Q (dfnrb1)               0.34       0.34 r
  a2b_sync/fb_or/Z (or02d2)                0.15       0.49 r
  a2b_sync/fb_reg/D (dfnrb1)               0.00       0.49 r
  data arrival time                                   0.49

  max_delay                               10.00      10.00
  library setup time                      -0.08       9.92
  data required time                                  9.92
  -----------------------------------------------------------
  data required time                                  9.92
  data arrival time                                  -0.49
  -----------------------------------------------------------
  slack (MET)                                        9.43
```

The non_sync_flops might result in a large collection, so the alternative would be to do set_max_delay with some very large number from the cdc clock, then the correct set_max_delay as an override.

```
# Or put large max delay first
set gray_flops [filter_collection [all_registers] "full_name =~ *_gray/*"]
foreach_in_collection cdcclk [get_clocks *_cdc] {
  &cmd set_max_delay 1000.0 -from $cdcclk
  &cmd set_max_delay [get_attribute $cdcclk period] -from $cdcclk -through
$gray_flops
}
```

Although not quite as "clean", this is probably the preferable implementation as it is likely to be much faster.

5 What about SI?

SI (signal integrity analysis) is a complicated beast, and every time I think I've covered all the bases I find another issue lurking somewhere.

I think this approach should handle SI effects correctly for interactions between clocks. The _cdc clocks are ideal and therefore do not have their insertion delays included, so their SI windows will not be strictly correct. However, since they are only used for checks between asynchronous domains, this should be irrelevant since the windows can be infinite anyway.

But the above code doesn't make the windows infinite. Instead of doing NO set_clock_groups for the _cdc clocks, we need to duplicate the STA set_clock_groups -async statement, but add "-allow_paths":

```
# Set them async but allow paths (for SI)
&cmd set_clock_groups -async -group {clka_cdc} -group {clkb_cdc} -allow_paths
```

Even with this change, SI effects within each _cdc clock domain are another matter. Each _cdc clock will see SI effects within its own domain using windows that ignore the clock tree. If the clock tree were perfect, this would not matter, since the edges and windows would move together. But real world clock tree skew introduces a risk that SI might miss a window/transition alignment - which could result in STA optimism.

I can't think of a really good way around this. You could get rid of this optimism (and get excess pessimism) by changing the physically_exclusive for each _cdc clock to be to all STA clocks except its STA twin, then set it and its twin as async. That would make for infinite windows from the STA twin to the _cdc clock. Unfortunately, it would also make for infinite windows between the totally bogus _cdc clock and its STA twin. And, you'd probably have to replace "STA twin" with "all STA clock in that clock group". Yech.

Or, you could create a set of _prop clocks (duplicating all the STA clocks just like for _cdc clocks) and use these to get infinite windows between the _prop clock and the _cdc clock. Another full set of clocks? Not very appealing. But, if you're only concentrating on a small number of crossings, this might be acceptable. Similarly, if you're fully automating this technique (a difficult task, as will be discussed in the next section), what's another line in the loop?

Fortunately, clock tree skew is generally small, so maybe we should just throw in a fudge factor when applying the set_max_delay, then double-check anything that gets close using get_timing_paths in SI mode.

6 Automating this algorithm

And now, let's discuss the issues involved in creating a fully automated solution.

To summarize, here are the steps required:

1. **Create the cdc clocks**
2. Set the cdc clocks unpropagated
3. Disable the cdc clock internal paths
4. **Replicate the STA physical/logical exclusive groups, substituting the cdc clocks**
5. **Create set_clock_groups -log commands for each normal STA group that has more than 1 member**
6. Set the cdc clocks physically exclusive from the normal STA clocks
7. Apply the set_max_delay

Steps 2,3,6 and 7 (not in bold) have already been shown in a fully automated, generic form.

Steps 1, 4 and 5 require more effort.

6.1 Creating the cdc clocks

Although creating the cdc clocks in the early examples looks simple, it is not quite so easy to automate. The problem is that you need all the information about the clock - it's period, waveform, master, master source, source, etc.

Unfortunately, these values are not readily available in any of the tools. There are two ways around this:

6.1.1 Create them simultaneously with the STA clocks using a wrapper script

One alternative is to do your clock creation using wrapper scripts. This is my preferred solution. This not only allows the cdc clocks to be created when the original STA clocks are created, it also allows you to squirrel-away other data about the clocks (waveform, master for generated clock, period, etc) into global data structures that can then help avoid unnecessary timing updates. This is what I use in my own flow.

6.1.2 Create them after the fact using a loop

The other option is to create them using a stand-alone loop. The catch here is that the information you need is not available via attributes. The only way that I know of to extract this information is from "report_clocks".

A technique for doing this is detailed in reference (2). The snag is, this is always tool-dependent. It works fine in PrimeTime, but I've never attempted it in DesignCompiler.

If you attempt this, keep in mind you cannot create a generated clock of a master clock that doesn't exist yet. Since generated clocks can themselves be masters, it is not sufficient to just create all the non-generated clocks, then the generated ones.

One way around this is to do something like this:

```
set clks [get_clocks *]
while {[sizeof_collection $clks] > 0} {
  set clk [index_collection $clks 0]
  if {[is_false [get_attribute $clk is_generated]]} {
    &cmd create_clock -add ...
    set clks [remove_from_collection $clks $clk]
  } else {
    # gen'd clock - determine master.  If the master clock doesn't exist yet,
    # skip this clock.
    set master [get_object_name [&master_of $clk]]
    if {[sizeof_collection [get_clocks -quiet $master_cdc]] > 0} {
      &cmd create_generated_clock ...
      set clks [remove_from_collection $clks $clk]
      ...
    }
  }
}
```

Another complication is generated clocks created by library models. report_clocks will return "*" as their master until after a timing update. Since we're trying to avoid unnecessary timing updates, this is a problem. One solution is to override these clocks with your own clocks (duplicating the div ratio, etc), but this opens up the danger of doing it incorrectly, or having the library model change and the sdc not be updated.

More detail on this technique is beyond the scope of this paper. To explain it fully would probably be a paper by itself. Most readers will probably either use a hand-coded approach or use the wrapper technique.

6.2 Replicating/modifying the STA clock groups for the cdc clocks

The STA clock group information has to be examined and translated as follows:

1. To duplicate the phys/log exclusive groups using _cdc clocks.
2. To make the _cdc version of each member of an STA clock group logically exclusive to all the other _cdc clocks of that group.

As with creating the _cdc clocks, these commands can be created and executed by hand, by using a wrapper, or fully automatically using a stand-alone loop.

Personally, I use a wrapper around set_clock_groups anyway, and I store the clock group information in an array (or several arrays), so I can implement this fairly easily using the wrapper approach.

It is also possible to do this by parsing the output of report_clocks -group, again using the technique from reference (2).

6.3 The bottom line

The bottom line is that there is no easy way to automate this as a standalone piece of code. This is primarily because so much information is required about the clocks and their relationships that is not available directly as attributes. I have managed to automate it, but only using a lot of features very specific to my flow.

7 How Synopsys Could Help

Much of the complexity of this approach arises from the fact that the set_max_delay command uses the clock insertion delay on the launch path. Providing an option "-no_clock_tree" would allow us to use set_max_delay directly, without the special ideal clocks. It would also make the set_max_delay command more generally useful, since there are few practical applications for it with the current functionality.

When combined with the "-allow_paths" option on set_clock_groups, this would make for a much more elegant solution.

I suppose the tricky bit is how to handle SI. For the purposes described in this paper, infinite windows are fine, but to make such an option generally useful would probably require that the clock insertion delay be calculated and used for SI analysis, then removed or cancelled to check the constraint.

Synopsys could also help simplify the automation by making all clock attributes available to tcl scripts (period, waveform, source, master, master source, etc).

8 Conclusion

We have examined the problem with leaving clock domain crossings untimed in the presence of gray-coded fifos and have seen that this can create undetected failures. We have explored some techniques for constraining these crossings. These techniques are effective, but are rather complex to implement on a large scale or in a generic way, particularly if one wants to avoid any SI optimism.

Still, the techniques can be used with today's tools and much of the complexity can be avoided if the user is only targeting particular crossings involving a small number of clocks.

The author hopes that this paper will stimulate discussion about this issue that may lead to cleaner, less complex techniques in the future.

9 Acknowledgements

The author would like to acknowledge the following individuals for their assistance:

Stuart Hecht, Independent ASIC Design Consultant, for his very detailed review of the early drafts of this paper which resulted in many improvements in both the paper and the underlying techniques.

Mark Sprague, Teradyne, for his detailed review on behalf of the SNUG technical commitee.

10 References

(1) My Favorite DC/PT Tcl Tricks
Paul Zimmer
Synopsys Users Group 2003 San Jose
(available at www.zimmerdesignservices.com)

(2) There's a better way to do it! Simple DC/PT tricks that can change your life
Paul Zimmer
Synopsys Users Group 2010 San Jose
(available at www.zimmerdesignservices.com)

(3) Simplifying Constraints By Using More Generated Clocks
Stuart Hecht [SJH Clear Consulting LLC]
Synopsys Users Group 2009 San Jose
(available at SNUG website)