Where have all the phases gone?
Using multiclock propagation in PrimeTime

Paul Zimmer

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

ABSTRACT

Primetime allows multiple clocks to propagate in parallel, allowing multiple operating modes to be timed in a single run. The paper will cover examples of when and how to use multiclock propagation as well as some new (2006.06 and 2006.12) features that make multiclock propagation much easier to use.

This paper has been updated since initial publication. Please visit my website www.zimmerdesignservices.com for the latest. –Paul Zimmer

Table of contents

1 Introduction - What is multiclock propagation and why is it useful.

Back in 2001, I presented a paper called "Complex Clocking Situations in PrimeTime" (ref [1]). In that paper, I discussed the issue of multiplexed clocks using the following example:

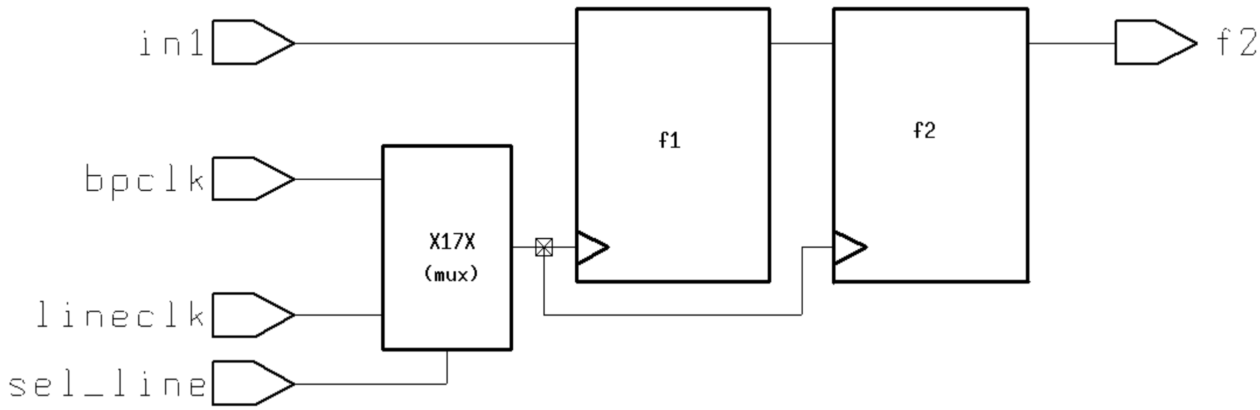


Figure 1-1

In those days, if you just created the two input clocks, only one would propagate, depending on the set_case_analysis value applied to sel_line. If no set_case_analysis was applied, the one that propagated was the last one declared. The only way to correctly time this circuit in those days was to run PT twice, using set_case_analysis to select a different clock in each case. Primetime did not at the time have any way to propagate multiple clocks on the same net in parallel.

This feature (multiclock propagation) was added shortly thereafter, but was turned off by default for the first few years. It is now on by default.

So, why use it? Well, for one thing, it reduces the number of PT runs required to tape out. With all the corners being run these days, cutting down on the number of base PT runs can save a lot of time and effort.

Perhaps a more important reason to use multiclock propagation is to get accurate noise analysis. Using multiple PT runs can sometimes lead to overly optimistic noise analysis, since some clocks that will be running on a real die might not be enabled in that PT run.

Another reason to use multiclock propagation is to learn the techniques so that they can be applied to synthesis and physical design tools in the future. Timing-driven synthesis and physical design results are only as good as the constraints, and multiclock propagation is the only way to let the tools "see" the whole optimization problem at once.

Also note that the alternative to multiclock propagation, using set_case_analysis, has its own problems. Synthesis and layout tools will sometimes build structures (using xor or an equivalent) that stop the case_value from propagating.

2 Using multiclock propagation to time a muxed output circuit

2.1 The circuit

Suppose a set of output pins will be shared between two interfaces, one of which runs at half the speed of the other. The output stage might look like this:

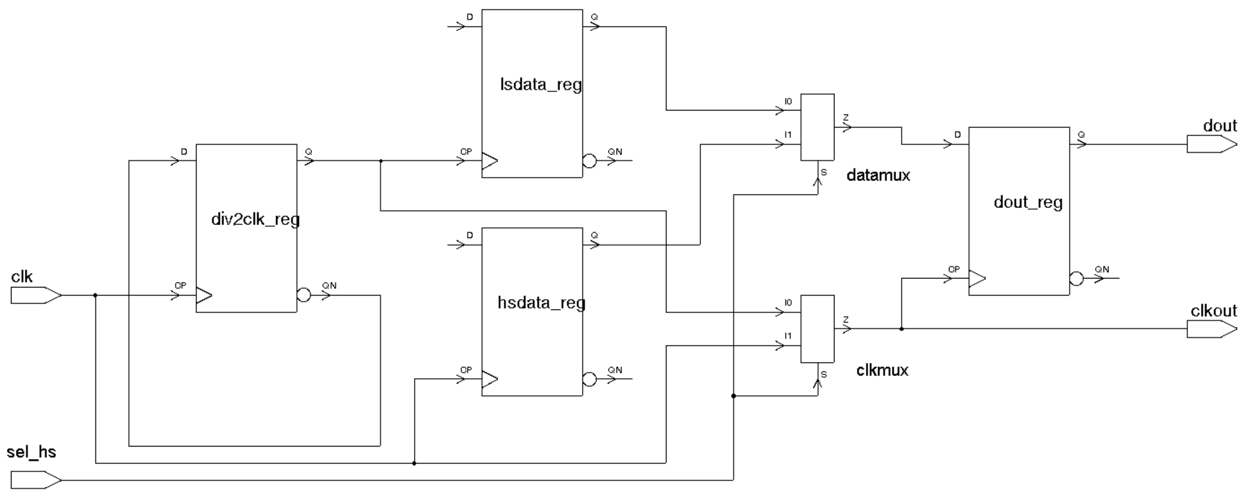


Figure 2-1

divclk_reg creates a divide-by 2 clock, which is then used to clock the low-speed interface logic (represented by lsdata_reg). The original, high-speed clock is used to clock the high-speed interface logic (represented by hsdata_reg). The outputs of these logic blocks, and their clocks, are then muxed into a final output flop and onto the output clock pin clkout to form a source-synchronous interface.

The clock and data flow look like this:

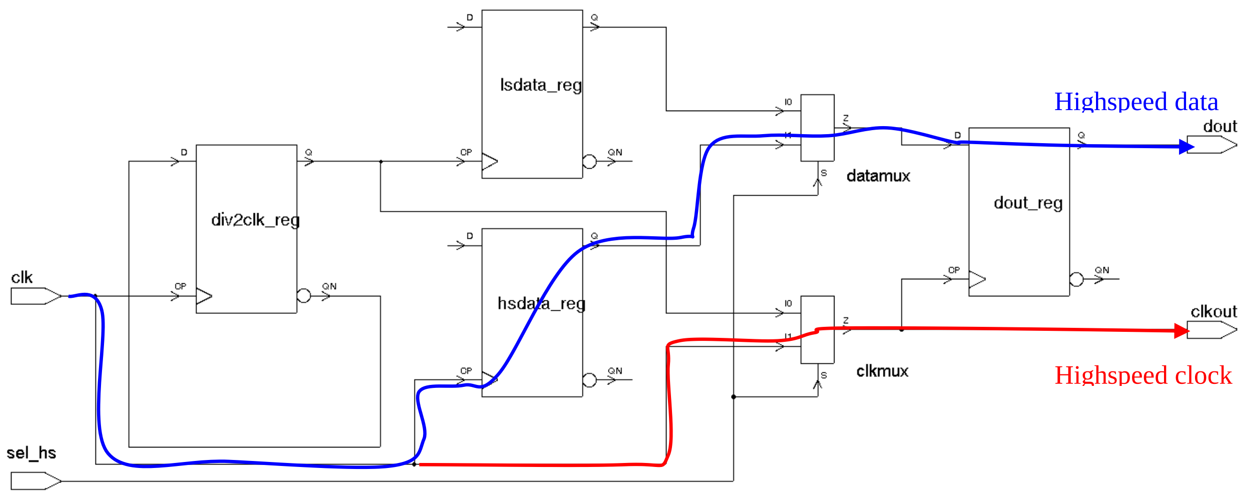


Figure 2-2

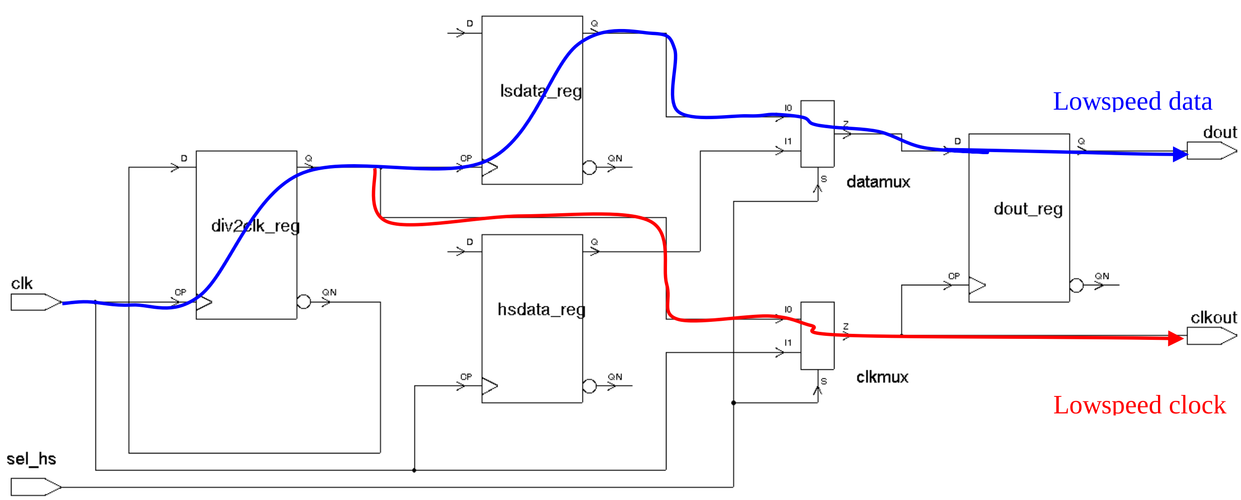


Figure 2-3

2.2 Timing using multiclock propagation

Rather than use set_case_analysis to control the muxes and using two PT runs, we can time both modes simultaneously using multiclock propagation.

First, I want to make sure multiclock propagation is on:

```
set timing_enable_multiple_clocks_per_reg true
```

I also turn off the switch that causes PrimeTime to create dummy input constraints:

```
set timing_input_port_default_clock false
```

And I use a new(er) switch that will cause all the clocks I create to be propagated:

```
set timing_all_clocks_propagated true
```

Now we create the input clock hsclk:

```
set hsperiod 1.0

create_clock -period $hsperiod [get_ports clk] -name hsclk
```

Then we create the divide-by 2 clock lsclk for low-speed mode:

```
create_generated_clock \
  -name lsclk \
  -source [get_attribute [get_clocks hsclk] sources] \
  -divide_by 2 \
  -master_clock hsclk \
  -add \
  [get_pins div2clk_reg/Q]
```

Next we need to create the two output clocks lsclkout and hsclkout. Lsclkout is slaved to lsclk and hsclkout is slaved to hsclk (this is standard technique for a source-sync output – see ref [1]). They are both created on the clkout port, so we must use –add:

```
create_generated_clock  \
  -name hsclkout \
  -source [get_attribute [get_clocks hsclk] sources] \
  -divide_by 1 \
  -master_clock hsclk \
  -add \
  [get_ports clkout]

create_generated_clock  \
  -name lsclkout \
  -source [get_attribute [get_clocks lsclk] sources] \
  -divide_by 1 \
  -master_clock lsclk \
  -add \
  [get_ports clkout]
```

The report_clock output then looks like this:

```
Attributes:
    p - Propagated clock
    G - Generated  clock
    I - Inactive   clock

Clock           Period   Waveform            Attrs     Sources
--------------------------------------------------------------------------------
-
hsclk             1.00   {0 0.5}             p         {clk}



Generated     Master         Generated      Master         Waveform
Clock         Source         Source         Clock          Modification
--------------------------------------------------------------------------------
-
hsclkout      clk            clkout         hsclk          div(1)
lsclk         clk            div2clk_reg/Q  hsclk          div(2)
lsclkout      div2clk_reg/Q  clkout         lsclk          div(1)
```

Set appropriate output delays relative to each output clock:

```
set_output_delay -min [expr -1 * 0.1] -clock hsclkout [get_ports dout]
set_output_delay -max 0.5 -clock hsclkout [get_ports dout]
set_output_delay -min [expr -1 * 0.2] -clock lsclkout [get_ports dout]
-add_delay
set_output_delay -max 1.2 -clock lsclkout [get_ports dout] -add_delay
```

2.3 Clock groups

Now let's look at the timing:

```
report_timing -to dout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by lsclk)
  Endpoint: dout (output port clocked by hsclkout)
  Path Group: hsclkout
  Path Type: max

  Point                                      Incr       Path
  --------------------------------------------------------------
  clock lsclk (rise edge)                    0.00       0.00
  clock network delay (propagated)           0.50       0.50
  dout_reg/CP (dfnrb1)                       0.00       0.50 r
  dout_reg/Q (dfnrb1)                        0.32       0.82 f
  dout (out)                                 0.00       0.82 f
  data arrival time                                     0.82

  clock hsclkout (rise edge)                 1.00       1.00
  clock network delay (propagated)           0.16       1.16
  output external delay                     -0.50       0.66
  data required time                                    0.66
  --------------------------------------------------------------
```

```
data required time                                      0.66
data arrival time                                      -0.82
---------------------------------------------------------------
slack (VIOLATED)                                       -0.16


Startpoint: dout_reg (rising edge-triggered flip-flop clocked by hsclk)
Endpoint: dout (output port clocked by lsclkout)
Path Group: lsclkout
Path Type: max

Point                                       Incr       Path
---------------------------------------------------------------
clock hsclk (rise edge)                     1.00       1.00
clock network delay (propagated)            0.16       1.16
dout_reg/CP (dfnrb1)                        0.00       1.16 r
dout_reg/Q (dfnrb1)                         0.32       1.48 f
dout (out)                                  0.00       1.48 f
data arrival time                                      1.48

clock lsclkout (rise edge)                  2.00       2.00
clock network delay (propagated)            0.50       2.50
output external delay                      -1.20       1.30
data required time                                     1.30
---------------------------------------------------------------
data required time                                     1.30
data arrival time                                     -1.48
---------------------------------------------------------------
slack (VIOLATED)                                      -0.18
```

We can see immediately that something is wrong. Data launched from dout_reg using lsclk is being captured by hsclkout, and data launched from dout_reg using hsclk is being captured by lsclk.

This is because PrimeTime assumes that all clocks should time against all other clocks unless we tell it otherwise. Interestingly, this is another issue addressed in that old 2001 paper (ref [1]) – managing all these cross-clock false paths when there are large numbers of clocks and generated clocks involved. Since the publication of that paper, the PrimeTime folks have given us a more elegant solution to this problem as well – set_clock_groups (it seems they *do* listen).

Set_clock_groups allows you to put your clocks into groups. All clocks within a group time against each other[1], but none of them time against any clocks in other groups.

We have two groups – hsclk and lsclk. We can use wildcarding to make this simple:

```
set_clock_groups -name muxed_out -logically_exclusive \
  -group [get_clocks "hs*"] \
  -group [get_clocks "ls*"]
```

Note the use of "-logically_exclusive". This is a new switch starting with 2006.06. It tells PrimeTime that timing paths between these clocks can be safely ignored (that's the exclusive

[1] Actually, the command says nothing about the clocks within a group, but since the default is to time against each other, the effect is that they time against each other – at least for simple examples.

part), but that both clocks *can* be physically present on the die at the same time. So, interactions between these clock groups for noise analysis purposes cannot be ignored. This differs from "-asynchronous" in that the noise timing windows are determined by the synchronous behavior of the clocks. We shall see an example of physically exclusive clock groups in the next section.

Note that I'm being a little bit pessimistic here. The low-speed and high-speed clocks *are* physically exclusive downstream of the clock mux. But since I expect that the bulk of the two clock trees are upstream of the clock mux and can thus interact, I don't try to explain this small piece of the problem to PrimeTime. It is possible to do this, but it's a little tricky and beyond the scope of what I'm trying to convey here.

Now our timing looks more reasonable:

```
report_timing -to dout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by hsclk)
  Endpoint: dout (output port clocked by hsclkout)
  Path Group: hsclkout
  Path Type: max

  Point                                    Incr       Path
  ---------------------------------------------------------------
  clock hsclk (rise edge)                  0.00       0.00
  clock network delay (propagated)         0.16       0.16
  dout_reg/CP (dfnrb1)                     0.00       0.16 r
  dout_reg/Q (dfnrb1)                      0.32       0.48 f
  dout (out)                               0.00       0.48 f
  data arrival time                                   0.48

  clock hsclkout (rise edge)               1.00       1.00
  clock network delay (propagated)         0.16       1.16
  output external delay                   -0.50       0.66
  data required time                                  0.66
  ---------------------------------------------------------------
  data required time                                  0.66
  data arrival time                                  -0.48
  ---------------------------------------------------------------
  slack (MET)                                         0.18


  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by lsclk)
  Endpoint: dout (output port clocked by lsclkout)
  Path Group: lsclkout
  Path Type: max

  Point                                    Incr       Path
  ---------------------------------------------------------------
  clock lsclk (rise edge)                  0.00       0.00
  clock network delay (propagated)         0.50       0.50
  dout_reg/CP (dfnrb1)                     0.00       0.50 r
  dout_reg/Q (dfnrb1)                      0.32       0.82 f
  dout (out)                               0.00       0.82 f
  data arrival time                                   0.82

  clock lsclkout (rise edge)               2.00       2.00
  clock network delay (propagated)         0.50       2.50
```

```
  output external delay                      -1.20       1.30
  data required time                                     1.30
  ---------------------------------------------------------------
  data required time                                     1.30
  data arrival time                                     -0.82
  ---------------------------------------------------------------
  slack (MET)                                            0.48
```

lsclk now times against lsclkout and hsclk times against hsclkout.

2.4 The promiscuous clock problem

But there's still a problem. Take a look at this timing report:

```
report_timing -to dout -delay min -group hsclkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by hsclk)
  Endpoint: dout (output port clocked by hsclkout)
  Path Group: hsclkout
  Path Type: min

  Point                                       Incr       Path
  ---------------------------------------------------------------
  clock hsclk (rise edge)                     0.00       0.00
  clock network delay (propagated)            0.16       0.16
  dout_reg/CP (dfnrb1)                        0.00       0.16 r
  dout_reg/Q (dfnrb1)                         0.32       0.48 r
  dout (out)                                  0.00       0.48 r
  data arrival time                                      0.48

  clock hsclkout (rise edge)                  0.00       0.00
  clock network delay (propagated)            0.50       0.50
  output external delay                       0.10       0.60
  data required time                                     0.60
  ---------------------------------------------------------------
  data required time                                     0.60
  data arrival time                                     -0.48
  ---------------------------------------------------------------
  slack (VIOLATED)                                      -0.12
```

Why is the clock insertion delay ("clock network delay (propagated)") of the capture clock so much larger than that of the launch clock? If we use "-path full_clock_expanded" the issue becomes apparent:

```
report_timing -to dout -delay min -path full_clock_expanded -input -group
hsclkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by hsclk)
  Endpoint: dout (output port clocked by hsclkout)
  Path Group: hsclkout
  Path Type: min

  Point                                         Incr       Path
  ---------------------------------------------------------------
  clock hsclk (rise edge)                       0.00       0.00
  clock source latency                          0.00       0.00
  clk (in)                                      0.00       0.00 r
  clkmux/I1 (mx02d0)                            0.00       0.00 r
  clkmux/Z (mx02d0)                             0.16       0.16 r
  dout_reg/CP (dfnrb1)                          0.00       0.16 r
  dout_reg/Q (dfnrb1)                           0.32       0.48 r
  dout (out)                                    0.00       0.48 r
  data arrival time                                        0.48

  clock hsclkout (rise edge)                    0.00       0.00
  clock hsclk (source latency)                  0.00       0.00
  clk (in)                                      0.00       0.00 r
  div2clk_reg/CP (dfnrb1)                       0.00       0.00 r
  div2clk_reg/Q (dfnrb1)                        0.32       0.32 r
  clkmux/I0 (mx02d0)                            0.00       0.32 r
  clkmux/Z (mx02d0)                             0.18       0.50 r
  clkout (out)                                  0.00       0.50 r
  output external delay                         0.10       0.60
  data required time                                       0.60
  ---------------------------------------------------------------
  data required time                                       0.60
  data arrival time                                       -0.48
  ---------------------------------------------------------------
  slack (VIOLATED)                                        -0.12
```

Why is the high-speed output clock hsclkout going through the clock divider?? This is the "promiscuous clock" problem I discussed in my 2006 paper "Getting DDR's Write" (ref [5]). PrimeTime will back up through any number of sequential logic elements to find the longest path. Back then, the only way around this at the time was to create "steering" clocks to force the clock to take the desired path.

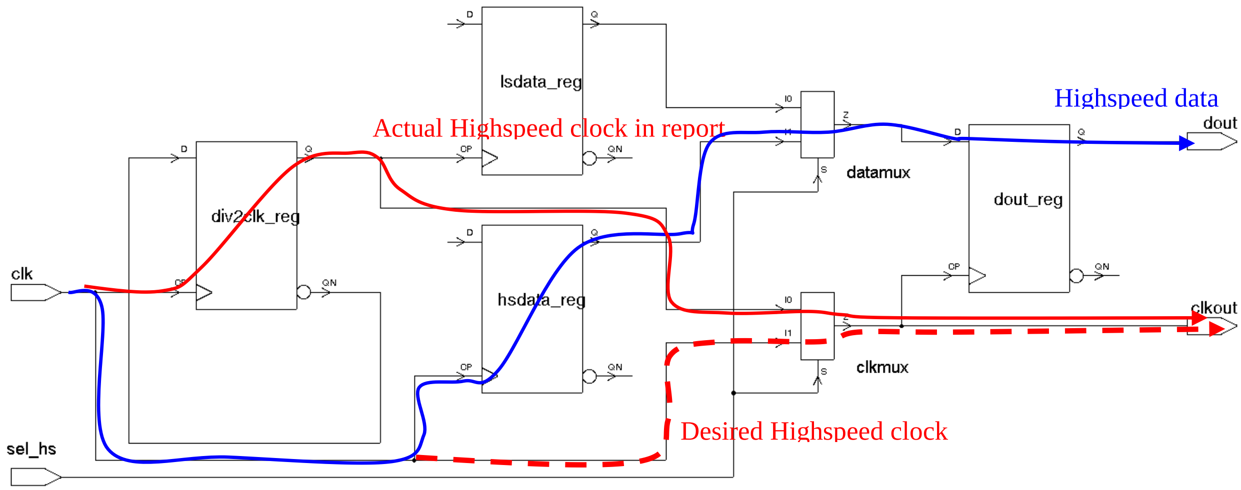


Figure 2-4

2.5 A new solution to this old problem – the –combinational switch

But the PrimeTime folks were listening (again)! In 2006.06, they have given us a new switch on create_generated_clock called "-combinational" that tells the tool that the generated clock is a divide-by 1, and the path from the source to the clock creation point is purely combinational.

This is true of both of our divide-by 1 output clocks (I have yet to come up with a divide_by 1 clock where this isn't true), so we can change the clock creation code to look like this:

```
create_generated_clock  \
  -name hsclkout \
  -source [get_attribute [get_clocks hsclk] sources] \
  -comb \
  -master_clock hsclk \
  -add \
  [get_ports clkout]

create_generated_clock  \
  -name lsclkout \
  -source [get_attribute [get_clocks lsclk] sources] \
  -comb \
  -master_clock lsclk \
  -add \
  [get_ports clkout]
```

And that fixes the problem:

```
report_timing -to dout -delay min -path full_clock_expanded -input -group
hsclkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by hsclk)
  Endpoint: dout (output port clocked by hsclkout)
  Path Group: hsclkout
  Path Type: min

  Point                                    Incr       Path
  ---------------------------------------------------------------
  clock hsclk (rise edge)                  0.00       0.00
  clock source latency                     0.00       0.00
  clk (in)                                 0.00       0.00 r
  clkmux/I1 (mx02d0)                       0.00       0.00 r
  clkmux/Z (mx02d0)                        0.16       0.16 r
  dout_reg/CP (dfnrb1)                     0.00       0.16 r
  dout_reg/Q (dfnrb1)                      0.32       0.48 r
  dout (out)                               0.00       0.48 r
  data arrival time                                   0.48

  clock hsclkout (rise edge)               0.00       0.00
  clock hsclk (source latency)             0.00       0.00
  clk (in)                                 0.00       0.00 r
  clkmux/I1 (mx02d0)                       0.00       0.00 r
  clkmux/Z (mx02d0)                        0.16       0.16 r
  clkout (out)                             0.00       0.16 r
  output external delay                    0.10       0.26
  data required time                                  0.26
  ---------------------------------------------------------------
  data required time                                  0.26
  data arrival time                                  -0.48
  ---------------------------------------------------------------
  slack (MET)                                         0.22
```

2.6 Avoiding the generated clock entirely: the –reference switch.

The divide-by 1 (now –comb) generated clock method has been used for years to time source-synchronous outputs. But there is another way – the "-reference" option on set_output_delay.

We create only the high-speed and low-speed clocks:

```
create_clock -period $hsperiod [get_ports clk] -name hsclk

create_generated_clock \
  -name lsclk \
  -source [get_attribute [get_clocks hsclk] sources] \
  -divide_by 2 \
  -master_clock hsclk \
  -add \
  [get_pins div2clk_reg/Q]
```

Then we do the set_output_delay statements using both "-clock" and "-reference_pin", with the reference pin being the clkout port:

```
set_output_delay -min [expr -1 * 0.1] -reference_pin [get_ports clkout] -clock
hsclk [get_ports dout]
set_output_delay -max 0.5 -reference_pin [get_ports clkout] -clock hsclk
[get_ports dout]

set_output_delay -max 1.2 -reference_pin [get_ports clkout] -clock lsclk
[get_ports dout] -add_delay
set_output_delay -min [expr -1 * 0.2] -reference_pin [get_ports clkout] -clock
lsclk [get_ports dout] -add_delay
```

And set the clock groups:

```
set_clock_groups -name muxed_out -logically_exclusive \
  -group [get_clocks "hs*"] \
  -group [get_clocks "ls*"]
```

This produces the same timing results. Re-running the previous trace, but with the group set to hsclk instead of hsclkout:

```
report_timing -to dout -delay min -path full_clock_expanded -input -group
hsclk

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by hsclk)
  Endpoint: dout (output port clocked by hsclk)
  Path Group: hsclk
  Path Type: min

  Point                                        Incr       Path
  ---------------------------------------------------------------
  clock hsclk (rise edge)                      0.00       0.00
  clock source latency                         0.00       0.00
  clk (in)                                     0.00       0.00 r
  clkmux/I1 (mx02d0)                           0.00       0.00 r
  clkmux/Z (mx02d0)                            0.16       0.16 r
  dout_reg/CP (dfnrb1)                         0.00       0.16 r
  dout_reg/Q (dfnrb1)                          0.32       0.48 r
  dout (out)                                   0.00       0.48 r
  data arrival time                                       0.48

  clock hsclk (rise edge)                      0.00       0.00
  clock source latency                         0.00       0.00
  clk (in)                                     0.00       0.00 r
  clkmux/I1 (mx02d0)                           0.00       0.00 r
  clkmux/Z (mx02d0)                            0.16       0.16 r
  clkout (out)                                 0.00       0.16 r
  output external delay                        0.10       0.26
  data required time                                      0.26
  ---------------------------------------------------------------
  data required time                                      0.26
  data arrival time                                      -0.48
  ---------------------------------------------------------------
  slack (MET)                                             0.22
```

This approach has the drawback, however, that it doesn't work in DesignCompiler. Of course, the "-comb" switch on create_generated_clock doesn't work in DesignCompiler, either.

The generated clock approach provides more flexibility for controlling the clock path. In simple cases (no muxing on the clock output), the "-reference_pin" might be simpler, but for more complex cases, you'll probably still want to use generated clocks.

3 Using multiclock propagation to time a shared set of input pins

3.1 The circuit

Now let's look at a circuit that shares pins between two different modes on the *input* side. Since these are inputs, there's no need for muxes. The signals are just routed to multiple places:

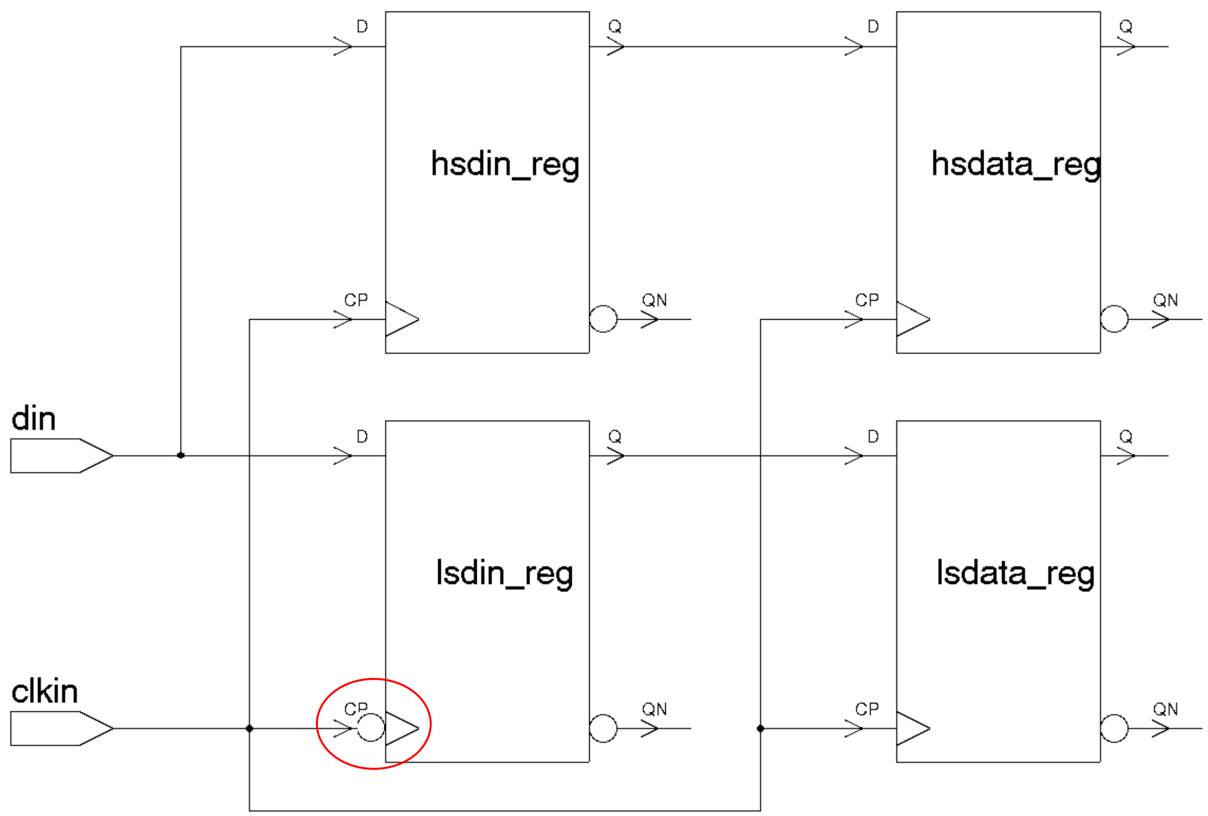


Figure 3-1

Notice that the high-speed interface (hsdin_reg) runs on rising clocks, while the low-speed interface (lsdin_reg) runs on falling clocks. The low-speed interface is center-clocked. Data is sourced on rising edges and captured on falling edges.

3.2 Timing using multiclock propagation

Using the –add switch, we can create both high-speed and low-speed clocks in the clkin pin:

```
set hsperiod 1.0
set lsperiod 10.0
create_clock -period $hsperiod [get_ports clkin] -name hsclkin
create_clock -period $lsperiod [get_ports clkin] -name lsclkin -add
```

Similarly, we apply input delays relative to both clocks to the data input pin din using – add_delay:

```
set_input_delay 0.1 -clock hsclkin -min [get_ports din]
set_input_delay 0.5 -clock hsclkin -max [get_ports din]
set_input_delay 0.7 -clock lsclkin -min [get_ports din] -add_delay
set_input_delay 2.5 -clock lsclkin -max [get_ports din] -add_delay
```

Notice that there's no "-clock_fall" on the lsclkin input delay. This is because the *source* launches data on rising edges. Which edge the logic captures on has no effect on the set_input_delay syntax.

As you would expect, this results in messy cross-clock paths between hsclkin and lsclkin, so we use set_clock_groups to disable these paths:

```
set_clock_groups -name muxed_in -physically_exclusive \
  -group [get_clocks "hs*"] \
  -group [get_clocks "ls*"]
```

This time, however, we use "-physically_exclusive" because these clocks *cannot* be present on the die at the same time, so they can be treated as mutually exclusive for both path analysis and noise analysis purposes.

This can get confusing, so here is a table that describes the settings and their implications for both noise and path timing:

Switch	Timing Paths	Noise Analysis	Timing Windows
-asynchronous	No	Yes	Infinite
-logically_exclusive	No	Yes	Per Waveforms
-physically_exclusive	No	No	N/A

3.3 Clocks everywhere!

But there's another problem lurking here. Recall that the pins are shared by two interfaces, one high-speed and one low-speed. Obviously the low-speed circuit is only required to run at the low-speed clock rate. But, because we have created and propagated both clocks down the same clock network, both clocks go to all flops. So, if we look at the timing to lsdin_reg, we will see it being checked against hsclk's period:

```
report_timing -to lsdin_reg/D -group hsclkin

  Startpoint: din (input port clocked by hsclkin)
  Endpoint: lsdin_reg (falling edge-triggered flip-flop clocked by hsclkin)
  Path Group: hsclkin
  Path Type: max

  Point                                          Incr       Path
  --------------------------------------------------------------
  clock hsclkin (rise edge)                      0.00       0.00
  clock network delay (propagated)               0.00       0.00
  input external delay                           0.50       0.50 f
  din (in)                                       0.00       0.50 f
  lsdin_reg/D (dfnfb1)                           0.00       0.50 f
  data arrival time                                         0.50

  clock hsclkin (fall edge)                      0.50       0.50
  clock network delay (propagated)               0.00       0.50
  lsdin_reg/CPN (dfnfb1)                                    0.50 f
  library setup time                            -0.12       0.38
  data required time                                        0.38
  --------------------------------------------------------------
  data required time                                        0.38
  data arrival time                                        -0.50
  --------------------------------------------------------------
  slack (VIOLATED)                                         -0.12
```

Since the low-speed circuit captures data on falling edges, there's no way this will ever work. Similarly, the flop-to-flop timing *inside* the low-speed block is also being constrained to operate at hsclk:

```
report_timing -to lsdata_reg/D -group hsclkin

  Startpoint: lsdin_reg (falling edge-triggered flip-flop clocked by hsclkin)
  Endpoint: lsdata_reg (rising edge-triggered flip-flop clocked by hsclkin)
  Path Group: hsclkin
  Path Type: max

  Point                                          Incr       Path
  ---------------------------------------------------------------
  clock hsclkin (fall edge)                      0.50       0.50
  clock network delay (propagated)               0.00       0.50
  lsdin_reg/CPN (dfnfb1)                         0.00       0.50 f
  lsdin_reg/Q (dfnfb1)                           0.32       0.82 r
  lsdata_reg/D (dfnrq1)                          0.00       0.82 r
  data arrival time                                         0.82

  clock hsclkin (rise edge)                      1.00       1.00
  clock network delay (propagated)               0.00       1.00
  lsdata_reg/CP (dfnrq1)                                    1.00 r
  library setup time                            -0.10       0.90
  data required time                                        0.90
  ---------------------------------------------------------------
  data required time                                        0.90
  data arrival time                                        -0.82
  ---------------------------------------------------------------
  slack (MET)                                               0.08
```

The problem is that both clocks go EVERYWHERE:

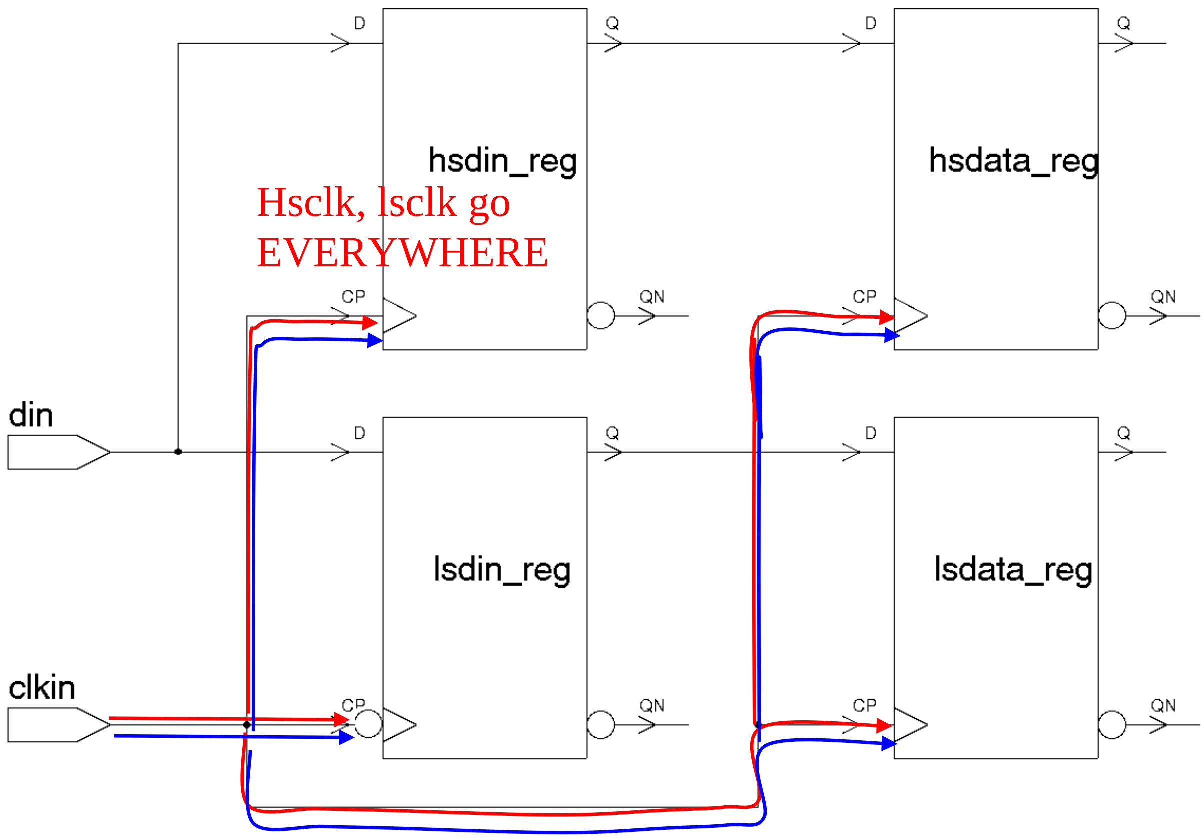


Figure 3-2

3.4 The new clock killer command

This is a nasty problem, and until recently there was no clean solution with PrimeTime. What we need is a way to "kill" clocks when they go places they aren't supposed to go. Starting with 2006.06, we have this capability. A "-stop_propagation" switch has been added to the set_clock_sense command. This switch will kill the clock on the specified point, *and anywhere downstream*.

In the example circuit, there's no handy place to put the set_clock_sense command where it will propagate downstream to the desired points, so we can just use the command directly on the flop clock pins:

```
set_clock_sense -stop_propagation -clock hsclkin [get_pins "ls*_reg/CP*"]
set_clock_sense -stop_propagation -clock lsclkin [get_pins "hs*_reg/CP*"]
```

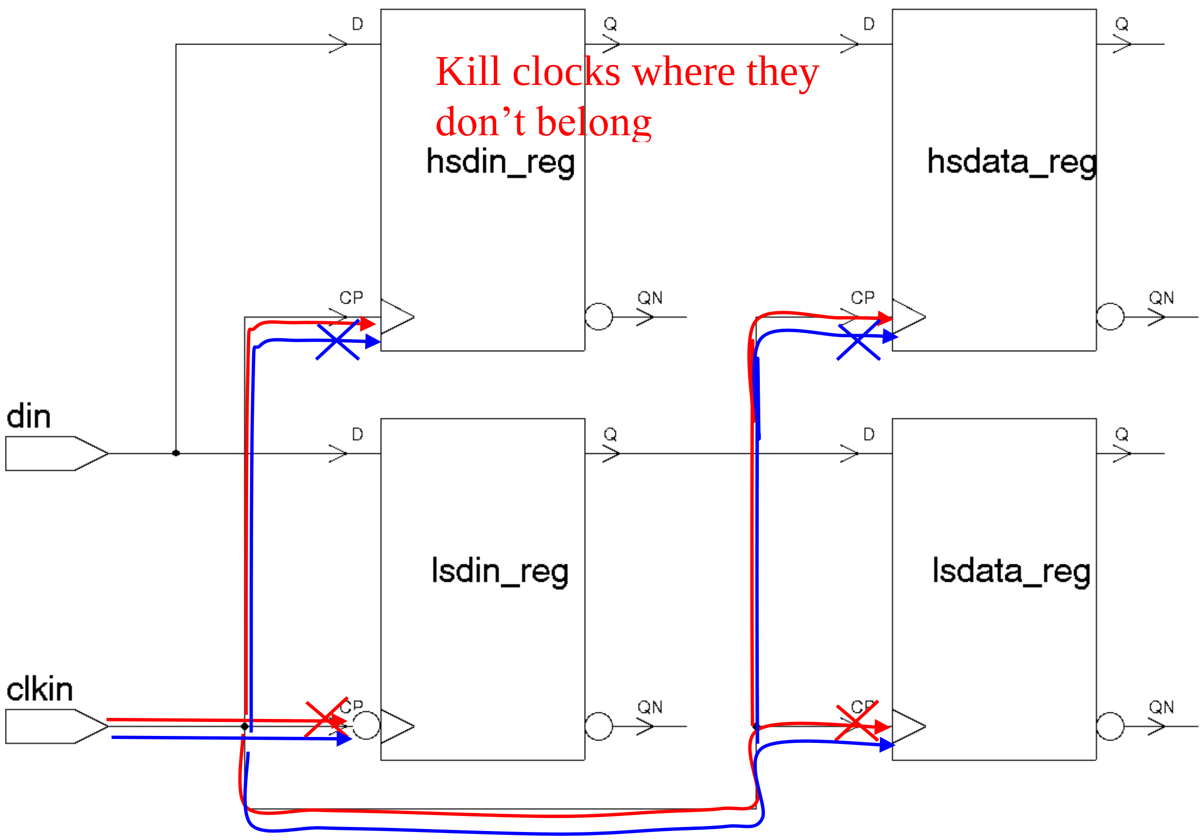


Figure 3-3

Now, when we run those reports, we get unconstrained paths:

```
report_timing -to lsdin_reg/D -group hsclkin


No constrained paths.

1
report_timing -to lsdata_reg/D -group hsclkin

No constrained paths.
```

Doing this on the flop clock pins works, but the switch is too new for me to be able to comment on performance. If this is too slow, you might have to find (or introduce) clock tree buffers that cover the required flops and put the set_clock_sense on these.

4 A more complex example – a multi-mode output circuit

4.1 The circuit

Now let's apply all this to something a bit more complex. Take a look at this circuit:

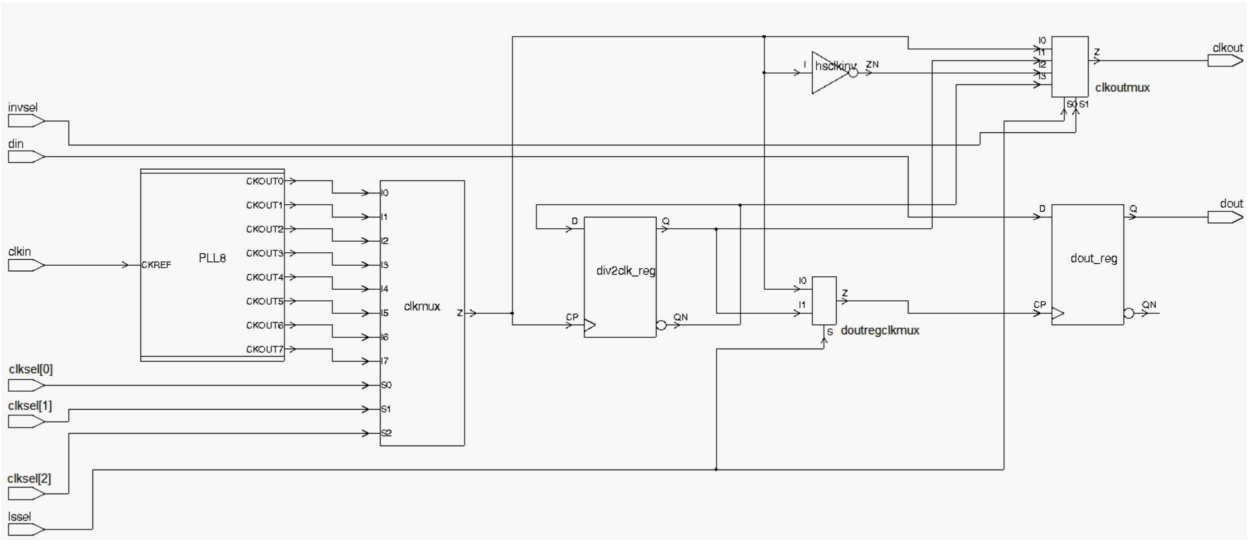


Figure 4-1

The circuit begins with a multi-tap PLL. This PLL has 8 taps, each 45 deg behind the previous tap. The outputs are fed into a mux. The mux output is then routed to a data output stage. The mux output is also, along with an inverted version, routed to the clock output pin. It is also routed to a divider, which can then be sent to the output flop (muxed with the pre-divider clock) and (along with an inverted version), can be sent to the clock output pin. So, you can toggle data with any of 8 phases, at full speed or half speed, and send it out with either its own clock or an inverted copy of its own clock.

Here's an example of the clock and data path for a circuit configured for high-speed operation on tap 5 with a non-inverted output clock:

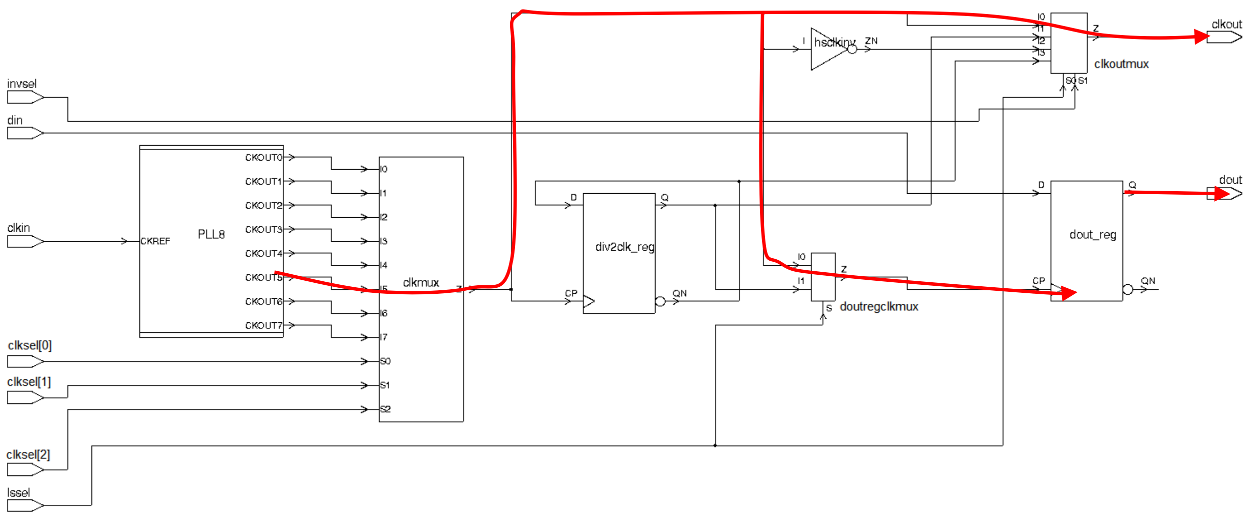


Figure 4-2

Here's an example of the clock and data path for a circuit configured for low-speed operation on tap 7 with an inverted output clock:

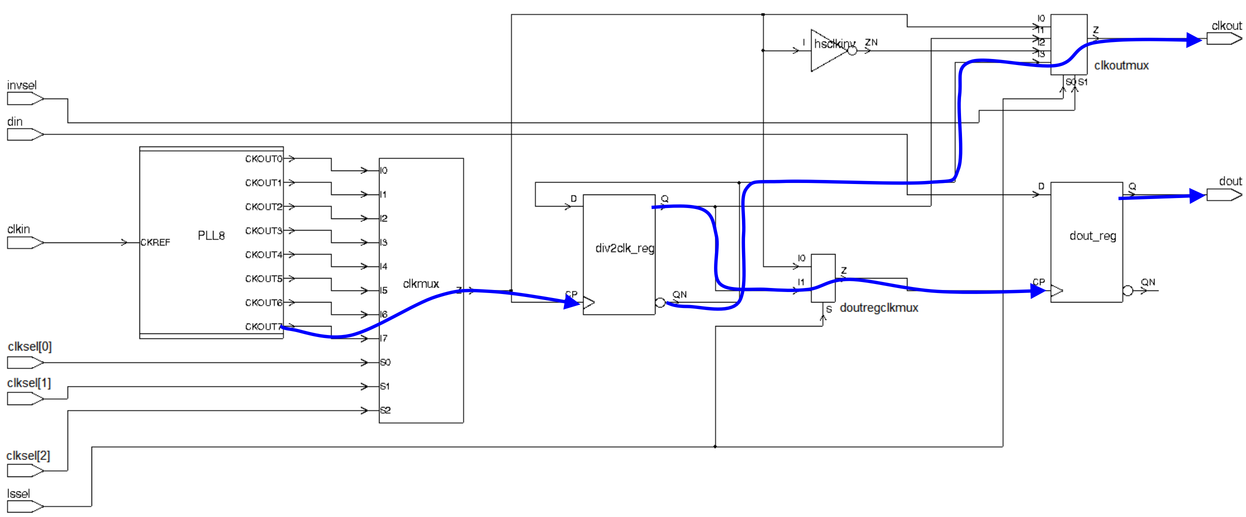


Figure 4-3

Depending on the environment the chip is used in, any one of these modes might be used. Also, there are several copies of this circuit on the die and each copy might be used differently in the same installation.

Even if you wanted to run all those corners, unless you run all combinations you might miss noise issues. So, let's do it all in one run with multiclock propagation.

4.2 The strategy

So, where do we begin? Well, each mode will have (potentially) different settings for the control bits clkset[2:0], lssel, and invsel. Rather than apply these settings directly using set_case_analysis, we're going to create clock paths for each mode that reflect its settings.

We control the clock paths by creating the correct generated clocks to force them to go where we want them to go.

The first thing we need to consider is the creation of the PLL output clocks. It would be possible to create the selected clock (reflecting that mode's clksel setting) for each mode directly on the correct PLL output. PLL taps that are used by more than one mode would have multiple clocks created on them, and those not used in any modes would have none. It would look something like this:

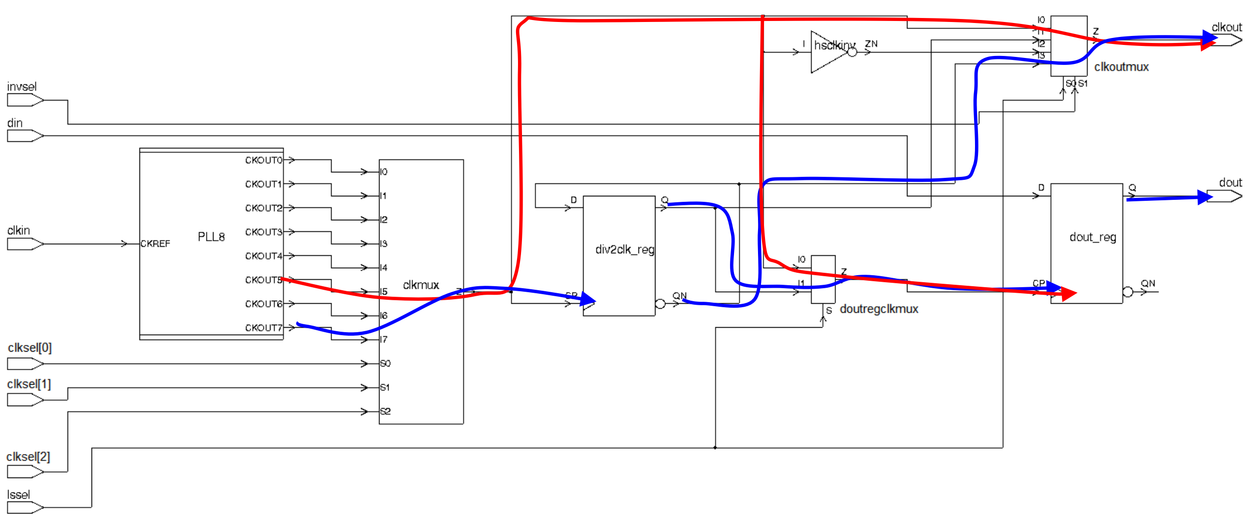


Figure 4-4

I don't like this approach for a couple of reasons. First, not creating the 'unused" clocks doesn't mean they don't exist on the real chip. For noise analysis purposes, I'd like all the PLL tap clocks to exist in every run. Second, the same PLL might be used for one or more other copies of this circuit. If I create a tap clock on the PLL for a particular mode in instance 1, it will leak into instances 2, 3, etc.

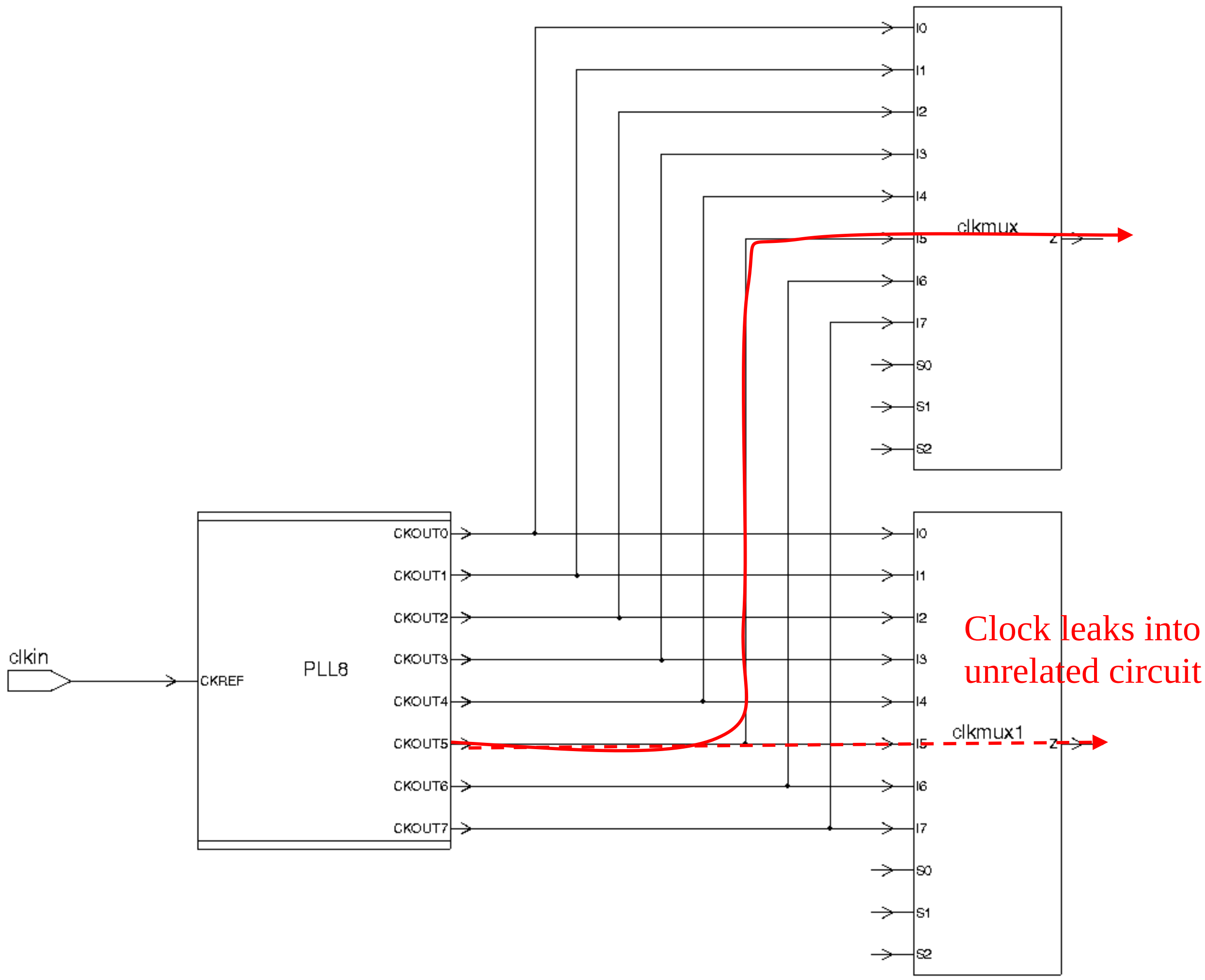


Figure 4-5

Instead, I prefer to create all the PLL tap clocks on their PLL output pins, then create divide-by 1 generated clocks at the mux output pin for each mode. This allows the PLL tap clocks to propagate everywhere they need to for noise anlaysis, but prevents mode clocks for any particular instance of the circuit from propagating to other instances (since each instance has its own clock mux).

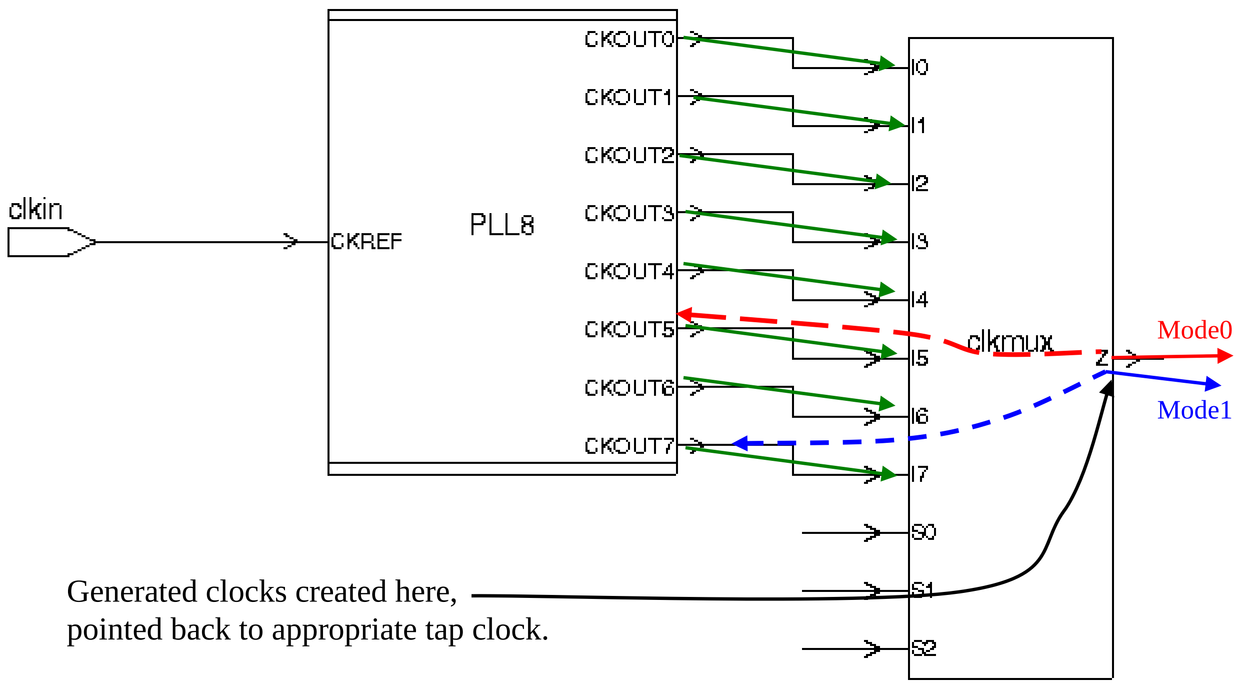


Figure 4-6

4.3 Creating the PLL tap clocks

OK, so let's create the PLL tap clocks. The "PLL" part of this is beyond the scope of this paper (see ref [4]), but suffice it to say that I prefer to create separate clocks on each output (rather than negative delays through the PLL). Given this, there are two possible ways of getting the phase shift. We could apply the phase shift as source latency, or we could create the clocks with the phase shift built into the waveform.

Either way will work, but the choice will affect which edge PrimeTime uses in timing checks. Since latency is ignored when figuring out which pair of launch/capture edges to use, a source latency technique will result in all checks using only the base (tap0) timing, which may result in incorrect checks and require multicycle paths. So, I will apply the shift by creating clocks with appropriate waveforms.

So, let's create the 8 phases using the waveform technique. The first time I attempted this, I did this:

```
for {set phase 0} {$phase < 8} {incr phase} {
  set offset [expr ($phase / 8) * $hsperiod]
  echo "phase: $phase ; offset: $offset"
  create_clock -name hsclk_p${phase} \
    -period $hsperiod \
    -waveform [list $offset [expr $offset + $hsperiod / 2] ] \
    [get_pins PLL8/CKOUT${phase}]
}
```

But this didn't work. The problem here is a bit subtle. Here's the report_clock output:

```
Attributes:
    p - Propagated clock
    G - Generated  clock
    I - Inactive   clock

Clock           Period   Waveform            Attrs     Sources
--------------------------------------------------------------------------------
-
hsclk_p0          4.00   {0 2}               p         {PLL8/CKOUT0}
hsclk_p1          4.00   {0 2}               p         {PLL8/CKOUT1}
hsclk_p2          4.00   {0 2}               p         {PLL8/CKOUT2}
hsclk_p3          4.00   {0 2}               p         {PLL8/CKOUT3}
hsclk_p4          4.00   {0 2}               p         {PLL8/CKOUT4}
hsclk_p5          4.00   {0 2}               p         {PLL8/CKOUT5}
hsclk_p6          4.00   {0 2}               p         {PLL8/CKOUT6}
hsclk_p7          4.00   {0 2}               p         {PLL8/CKOUT7}
```

All the clocks are the same! What's going on here?

Here's the echo output:

```
phase: 0 ; offset: 0.0
phase: 1 ; offset: 0.0
phase: 2 ; offset: 0.0
phase: 3 ; offset: 0.0
phase: 4 ; offset: 0.0
phase: 5 ; offset: 0.0
phase: 6 ; offset: 0.0
phase: 7 ; offset: 0.0
```

Why's the offset always zero?

Well…. Tcl has variable typing going on under the hood. The "for" loop creates "phase" as an integer. And an integer divided by an integer ("expr $phase / 8") results in a (truncated) integer. So, for phases of 0-7, the expression "$phase / 8" always results in zero:

```
pt_shell> echo [expr 7 / 8]
0
```

But if I force the dividend to be a float, I get the correct answer:

```
pt_shell> echo [expr 7.0 / 8]
0.875
```

Or, for the purist:

```
pt_shell> echo [expr double (7) / 8]
0.875
```

Since I'm using $phase as an integer to find the pin in "[get_pins PLL8/CKOUT${phase}]", I don't want to change the "for" loop index variable (phase) to a float. But I can force it to be a float just for the divide by using "double":

```
for {set phase 0} {$phase < 8} {incr phase} {
  set offset [expr (double($phase) / 8) * $hsperiod]
  echo "phase: $phase ; offset: $offset"
  create_clock -name hsclk_p${phase} \
    -period $hsperiod \
    -waveform [list $offset [expr $offset + $hsperiod / 2] ] \
    [get_pins PLL8/CKOUT${phase}]
}
```

This works. Here's the report_clock output:

```
Attributes:
    p - Propagated clock
    G - Generated  clock
    I - Inactive   clock

Clock          Period   Waveform            Attrs     Sources
--------------------------------------------------------------------------------
-
hsclk_p0         4.00   {0 2}               p         {PLL8/CKOUT0}
hsclk_p1         4.00   {0.5 2.5}           p         {PLL8/CKOUT1}
hsclk_p2         4.00   {1 3}               p         {PLL8/CKOUT2}
hsclk_p3         4.00   {1.5 3.5}           p         {PLL8/CKOUT3}
hsclk_p4         4.00   {2 4}               p         {PLL8/CKOUT4}
hsclk_p5         4.00   {2.5 4.5}           p         {PLL8/CKOUT5}
hsclk_p6         4.00   {3 5}               p         {PLL8/CKOUT6}
hsclk_p7         4.00   {3.5 5.5}           p         {PLL8/CKOUT7}
```

Note that "double" is not the only way to force this. You could also do this:

```
for {set phase 0} {$phase < 8} {incr phase} {
  set offset [expr ($phase / 8.0) * $hsperiod]
  echo "phase: $phase ; offset: $offset"
  create_clock -name hsclk_p${phase} \
    -period $hsperiod \
    -waveform [list $offset [expr $offset + $hsperiod / 2] ] \
    [get_pins PLL8/CKOUT${phase}]
}
```

The "8.0" will have the effect of forcing the term "$phase / 8.0" to be a float. Fun with Tcl…

Having created all the PLL tap clocks with multiclock prop on, what clocks do you suppose are now going through the clkmux/Z output pin? We can see using the new "clocks" attribute:

```
pt_shell> get_attribute [get_pins clkmux/Z] clocks
{"hsclk_p6", "hsclk_p7", "hsclk_p5", "hsclk_p4", "hsclk_p0", "hsclk_p3",
"hsclk_p1", "hsclk_p2"}
```

And they all go through to the divider reg clock pin as well:

```
pt_shell> get_attribute [get_pins div2clk_reg/CP] clocks
{"hsclk_p6", "hsclk_p7", "hsclk_p5", "hsclk_p4", "hsclk_p0", "hsclk_p3",
"hsclk_p1", "hsclk_p2"}
```

You can also see this using the new get_clock_network_objects command:

```
pt_shell> get_clock_network_objects -type pin hsclk_p0
{"clkoutmux/I0", "clkmux/Z", "dout_reg/CP", "clkoutmux/I2", "hsclkinv/ZN",
"clkoutmux/Z", "clkout", "doutregclkmux/I0", "doutregclkmux/Z",
"div2clk_reg/CP", "clkmux/I0", "hsclkinv/I"}
```

Note: For more info on the clocks attribute, and other tips on tracing clock paths, see solvnet article https://solvnet.synopsys.com/retrieve/009401.html.

Which means, at least for the moment, they all go everywhere. But my next step will be to create divide-by 1 generated clocks on the clkmux output pin. This will block the PLL tap phases from going any further.

4.4 High-speed Modes

4.4.1 Mode H1

Let's define a mode H1 (for high-speed 1), with the following setttings:

```
set modeH1(clksel_setting) 5
set modeH1(invsel_setting) 0
set modeH1(lssel_setting) 0
```

This means I want to use tap 5, use the highspeed path, and I want the output clock to be uninverted.

Since the clksel setting is 5, I want to create a divide-by 1 generated clock on pin clkmux/Z whose master is hsclk_p5 and whose source is PLL8/CKOUT5 (the source pin of clock hsclk_p5). I can do this with the following code:

```
create_generated_clock  \
  -name modeH1clk \
  -source [get_attribute [get_clocks hsclk_p${modeH1(clksel_setting)}]
sources] \
  -comb \
  -master_clock hsclk_p${modeH1(clksel_setting)} \
  -add \
  [get_pins clkmux/Z]
```

A couple of comments on this. First, I have concatenated "hsclk_p" with the value of the variable $modeH1(clksel_setting) to form the master clock name. I have used this as the "-master_clock" argument, and, via [get_attribute [get_clocks …] sources] to get the "-source argument" as well.

Second, notice that I have used "-comb" instead of '-divide_by 1". This is a divide-by 1 clock with a combinational path from its source, so using "-comb" will avoid some of the promiscuous clock problems discussed earlier.

We can see what all this results in by using my "&cmd" proc (see ref [3]):

```
remove_clock modeH1clk
&cmd create_generated_clock  \
  -name modeH1clk \
  -source [get_attribute [get_clocks hsclk_p${modeH1(clksel_setting)}]
sources] \
  -comb \
  -master_clock hsclk_p${modeH1(clksel_setting)} \
  -add \
  [get_pins clkmux/Z]
```

Which results in:

```
Doing command: create_generated_clock -name modeH1clk -source { PLL8/CKOUT5 }
-comb -master_clock hsclk_p5 -add { clkmux/Z }
```

My new clock is called "modeH1clk".

Now let's revisit those clock attribute values. All the PLL tap clocks still exist on the clkmux/Z pin:

```
pt_shell> get_attribute [get_pins clkmux/Z] clocks
{"hsclk_p0", "hsclk_p1", "hsclk_p2", "modeH1clk", "hsclk_p4", "hsclk_p3",
"hsclk_p5", "hsclk_p6", "hsclk_p7"}
```

But they stop there. If we look at the divider flop clock pin downstream, for example, we now have only modeH1clk:

```
pt_shell> get_attribute [get_pins div2clk_reg/CP] clocks
{"modeH1clk"}
```

And get_clock_network_objects now shows only the clkmux input and output pins:

```
pt_shell> get_clock_network_objects -type pin hsclk_p0
{"clkmux/Z", "clkmux/I0"}
```

The new generated clock has blocked the pll tap clocks from propagating beyond its creation point. This is a "feature" (or maybe just a property) of generated clocks (and non-generated clocks, for that matter). When you create a generated clock on a point, it blocks propagation of all other clocks through that point.

4.4.2 The HS output clock

Since our output is source synchronous, we need to create a divide-by 1 generated clock on the clock output port clkout (see [1]). But there are two paths from the clkmux to the clkout port – one through the inverter hsclkinv, and one around it. How will PT know which to use?

And here's where we come to a fork in the road. The first draft of this paper, prepared with version 2006.06-SP2, had a long explanation of why PT didn't get this right and how to get around it. But then 2006.12 arrived, and, lo and behold, they fixed it!

I don't wish to belabor you with now-irrelevant details related to now obsolete tool versions, but I think some background is in order here for those long-time PT users to fully appreciate this change.

In all versions prior to 2006.12, PT ignored edge inversion in the path to a generated clock. This is best illustrated with an example.

Using 2006.06-SP2, suppose we were to just create the output clock like this:

```
create_generated_clock  \
  -name modeH1clkout \
  -source [get_attribute [get_clocks modeH1clk] sources] \
  -comb \
  -master_clock modeH1clk \
  -add \
  [get_ports clkout]
```

And apply some output delays:

```
set_output_delay -min [expr -1 * 0.1] -clock modeH1clkout [get_ports dout]
-add_delay
set_output_delay -max 0.5 -clock modeH1clkout [get_ports dout] -add_delay
```

I use "-add_delay" on all set_output_delay commands because I will be applying output delays relative to multiple clocks and I don't want to keep track of which one goes first.

Put extra delay on the inverter to make it stand out in the timing reports:

```
set_annotated_delay 1.0 -cell -from hsclkinv/I -to hsclkinv/ZN
```

Here's the max timing report:

```
report_timing -delay max -to dout -path full_clock_expanded -input

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by modeH1clk)
  Endpoint: dout (output port clocked by modeH1clkout)
  Path Group: modeH1clkout
  Path Type: max

  Point                                       Incr       Path
  ---------------------------------------------------------------
  clock modeH1clk (rise edge)                 2.50       2.50
  clock hsclk_p5 (source latency)             0.00       2.50
  PLL8/CKOUT5 (DUMMYPLL8)                     0.00       2.50 r
  clkmux/I5 (mx08d1)                          0.00       2.50 r
  clkmux/Z (mx08d1) (gclock source)           0.62       3.12 r
  doutregclkmux/I0 (mx02d0)                   0.00       3.12 r
  doutregclkmux/Z (mx02d0)                    0.22       3.34 r
  dout_reg/CP (dfnrb1)                        0.00       3.34 r
  dout_reg/Q (dfnrb1)                         0.32       3.66 f
  dout (out)                                  0.00       3.66 f
  data arrival time                                      3.66

  clock modeH1clkout (rise edge)              6.50       6.50
  clock hsclk_p5 (source latency)             0.00       6.50
  PLL8/CKOUT5 (DUMMYPLL8)                     0.00       6.50 r
  clkmux/I5 (mx08d1)                          0.00       6.50 r
  clkmux/Z (mx08d1) (gclock source)           0.62       7.12 r
  clkoutmux/I0 (mx04d0)                       0.00       7.12 r
  clkoutmux/Z (mx04d0)                        0.28       7.40 r
  clkout (out)                                0.00       7.40 r
  output external delay                      -0.50       6.90
  data required time                                     6.90
  ---------------------------------------------------------------
  data required time                                     6.90
  data arrival time                                     -3.66
  ---------------------------------------------------------------
  slack (MET)                                            3.24
```

That looks OK. Now look what we get for a min timing report (in 2006.06-SP2):

```
report_timing -delay min -to dout -path full_clock_expanded -input

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by modeH1clk)
  Endpoint: dout (output port clocked by modeH1clkout)
  Path Group: modeH1clkout
  Path Type: min

  Point                                        Incr       Path
  ---------------------------------------------------------------
  clock modeH1clk (rise edge)                  2.50       2.50
  clock hsclk_p5 (source latency)              0.00       2.50
  PLL8/CKOUT5 (DUMMYPLL8)                      0.00       2.50 r
  clkmux/I5 (mx08d1)                           0.00       2.50 r
  clkmux/Z (mx08d1) (gclock source)            0.62       3.12 r
  doutregclkmux/I0 (mx02d0)                    0.00       3.12 r
  doutregclkmux/Z (mx02d0)                     0.20       3.32 r
  dout_reg/CP (dfnrb1)                         0.00       3.32 r
  dout_reg/Q (dfnrb1)                          0.32       3.64 r
  dout (out)                                   0.00       3.64 r
  data arrival time                                       3.64

  clock modeH1clkout (rise edge)               2.50       2.50
  clock hsclk_p5 (source latency)              0.00       2.50
  PLL8/CKOUT5 (DUMMYPLL8)                      0.00       2.50 f
  clkmux/I5 (mx08d1)                           0.00       2.50 f
  clkmux/Z (mx08d1) (gclock source)            0.59       3.09 f
  hsclkinv/I (inv0d0)                          0.00       3.09 f
  hsclkinv/ZN (inv0d0)                         1.00 *     4.09 r
  clkoutmux/I2 (mx04d0)                        0.00       4.09 r
  clkoutmux/Z (mx04d0)                         0.26       4.35 r
  clkout (out)                                 0.00       4.35 r
  output external delay                        0.10       4.45
  data required time                                      4.45
  ---------------------------------------------------------------
  data required time                                      4.45
  data arrival time                                      -3.64
  ---------------------------------------------------------------
  slack (VIOLATED)                                       -0.82
```

You can see that the max capture path bypassed the inverter, and the min capture path went through it. We have the classic “promiscuous clock” problem.

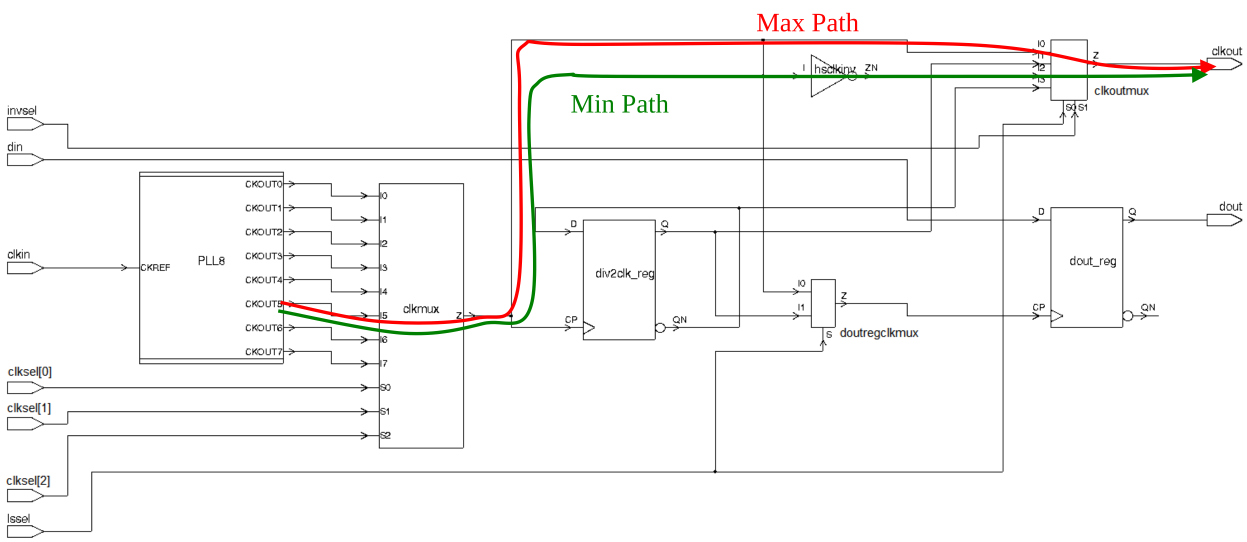


Figure 4-7

But there’s another problem with this trace. Notice that the timing check is rise-to-rise off the same edge from PLL8/CKOUT5, but it’s using an inverting clock path. If it’s using an inverted capture path, it should have been 2.50 vs 0.50. So, the check would have been wrong even if the non-inverting path didn’t exist.

```
clock modeH1clkout (rise edge)          2.50        2.50
clock hsclk_p5 (source latency)         0.00        2.50
PLL8/CKOUT5 (DUMMYPLL8)                 0.00        2.50 f
clkmux/I5 (mx08d1)                      0.00        2.50 f
clkmux/Z (mx08d1) (gclock source)       0.59        3.09 f
hsclkinv/I (inv0d0)                     0.00        3.09 f
hsclkinv/ZN (inv0d0)                    1.00 *      4.09 r
clkoutmux/I2 (mx04d0)                   0.00        4.09 r
clkoutmux/Z (mx04d0)                    0.26        4.35 r
clkout (out)                            0.00        4.35 r
output external delay                   0.10        4.45
data required time                                  4.45
-----------------------------------------------------------------
data required time                                  4.45
data arrival time                                  -3.64
-----------------------------------------------------------------
slack (VIOLATED)                                   -0.82
```

The 2.50 time corresponds to a *rising* edge at the PLL (tap 5 at 500ps/tap). Yet it shows up as a *falling* edge at the PLL output. So, all the subsequent times are bogus – they’re the wrong edge!

So, not only did the clock take the wrong path (through the inverter), the launch time of the path in inconsistent with the path itself. The bottom line was that, while PT would handle clock inversions in register-to-registers paths just fine, it didn't do this correctly for inversions in generated clock paths.

Until now.

Let's run that same min report using 2006.12:

```
report_timing -delay min -to dout -path full_clock_expanded -input

****************************************
Report : timing
        -path_type full_clock_expanded
        -delay_type min
        -input_pins
        -max_paths 1
Design : muxed_phase
Version: Z-2006.12
****************************************

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by modeH1clk)
  Endpoint: dout (output port clocked by modeH1clkout)
  Path Group: modeH1clkout
  Path Type: min

  Point                                     Incr       Path
  ---------------------------------------------------------------
  clock modeH1clk (rise edge)               2.50       2.50
  clock hsclk_p5 (source latency)           0.00       2.50
  PLL8/CKOUT5 (DUMMYPLL8)                   0.00       2.50 r
  clkmux/I5 (mx08d1)                        0.00       2.50 r
  clkmux/Z (mx08d1) (gclock source)         0.62       3.12 r
  doutregclkmux/I0 (mx02d0)                 0.00       3.12 r
  doutregclkmux/Z (mx02d0)                  0.20       3.32 r
  dout_reg/CP (dfnrb1)                      0.00       3.32 r
  dout_reg/Q (dfnrb1)                       0.32       3.64 r
  dout (out)                                0.00       3.64 r
  data arrival time                                    3.64

  clock modeH1clkout (rise edge)            2.50       2.50
  clock hsclk_p5 (source latency)           0.00       2.50
  PLL8/CKOUT5 (DUMMYPLL8)                   0.00       2.50 r
  clkmux/I5 (mx08d1)                        0.00       2.50 r
  clkmux/Z (mx08d1) (gclock source)         0.62       3.12 r
  clkoutmux/I0 (mx04d0)                     0.00       3.12 r
  clkoutmux/Z (mx04d0)                      0.30       3.42 r
  clkout (out)                              0.00       3.42 r
  output external delay                     0.10       3.52
  data required time                                   3.52
  ---------------------------------------------------------------
  data required time                                   3.52
  data arrival time                                   -3.64
  ---------------------------------------------------------------
  slack (MET)                                          0.12
```

The capture path no longer goes through the inverter at all! Which is correct, since we created a divide-by 1 ("-comb") generated clock and *didn't* use the –invert switch (an example using the -invert switch is coming up next). And, the launch and capture times are both correct. It all just works!

So, PT now understands that a divide-by 1, non-inverting clock should take a non-inverting path. Well, what happens if there isn't one?

Try this:

```
remove_clock modeH1clkout
disconnect_net [get_net hsclk] [get_pins clkoutmux/I0]
create_generated_clock  \
  -name modeH1clkout \
  -source [get_attribute [get_clocks modeH1clk] sources] \
  -comb \
  -master_clock modeH1clk \
  -add \
  [get_ports clkout]
```

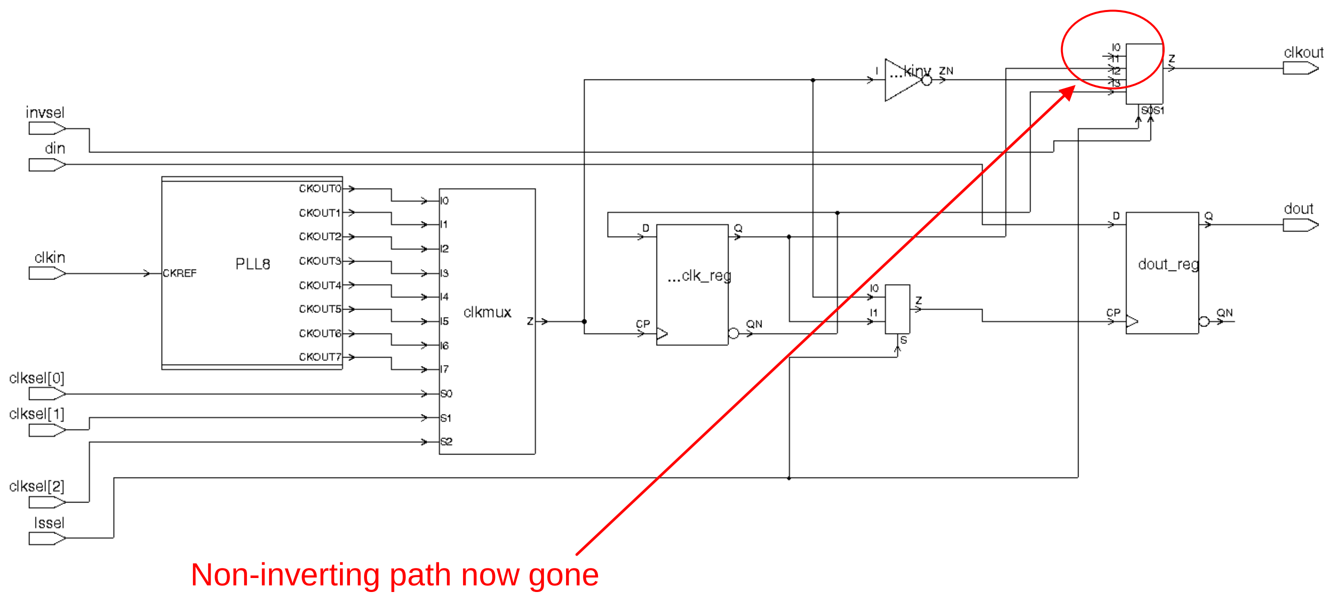


Figure 4-8

So, we disconnected the non-inverting path, leaving only the inverting path, but then created a non-inverting –comb generated clock. What will PT do now?

If you do update_timing, you get this:

```
Warning: Generated clock 'modeH1clkout' 'rise_edge' is not satisfiable; zero
source latency will be used.
 (UITE-461)
Warning: Generated clock 'modeH1clkout' 'fall_edge' is not satisfiable; zero
source latency will be used.
 (UITE-461)
```

UITE-461 is a new PT warning indicating exactly this problem!

This is an important point to understand. UITE-461 indicates there's something *wrong* with your clock creation commands! If you run an old script with 2006.12 and get UITE-461, then the script was broken before and your previous timing results were probably wrong! ***Do not ignore UITE-461.***

For a detailed explanation of how generated clock latencies were derived in various version of the tool, please see solvnet article *https://solvnet.synopsys.com/retrieve/015752.html*

4.4.3 Mode H2

So, now lets define a high-speed mode setting that *does* have invsel on:

```
set modeH2(clksel_setting) 3
set modeH2(invsel_setting) 1
set modeH2(lssel_setting) 0
```

As with H1, we create the clkmux clock:

```
# First, the clkmux clock
create_generated_clock  \
  -name modeH2clk \
  -source [get_attribute [get_clocks hsclk_p${modeH2(clksel_setting)}]
sources] \
  -comb \
  -master_clock hsclk_p${modeH2(clksel_setting)} \
  -add \
  [get_pins clkmux/Z]
```

To keep things simple, the H1 mode code above ignored the invsel_setting value and just created a non-inverting output clock. For H2, let's expand the code to create the inverted or non-inverted version based on the invsel_setting value.

We could use two different create_generated_clock statements, controlled by an "if". But I like to be sure that the *only* difference is the –invert switch, so I'm going to do it a little differently. I'll use the "if" to create a string variable containing the –invert switch (or null, if invsel_setting is false), then use "eval" to make TCL expand the variable before evaluating the statement:

```
  if {$modeH2(invsel_setting) == 1} {
    set invertarg {-invert }
  } else {
    set invertarg {}
  }
  # Create the output clock with the correct -invert setting
  eval create_generated_clock  \
    -name modeH2clkout \
    -source [get_attribute [get_clocks modeH2clk] sources] \
    -comb \
    -master_clock modeH2clk \
    -add \
    $invertarg \
    [get_ports clkout]
```

Apply the output delays:

```
set_output_delay -min [expr -1 * 0.15] -clock modeH2clkout [get_ports dout]
-add_delay
set_output_delay -max 0.7 -clock modeH2clkout [get_ports dout]  -add_delay
```

Since we now have multiple clocks, we need set_clock_groups to keep them separated:

```
# Separate the modes
set_clock_groups -name muxed_out -physically_exclusive \
  -group [get_clocks modeH1*] \
  -group [get_clocks modeH2*]
```

Now when we run that same report for modeH2, we can see the inverted capture path:

```
report_timing -delay min -to dout -path full_clock_expanded -input -group
modeH2clkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by modeH2clk)
  Endpoint: dout (output port clocked by modeH2clkout)
  Path Group: modeH2clkout
  Path Type: min

  Point                                         Incr       Path
  ---------------------------------------------------------------
  clock modeH2clk (rise edge)                   5.50       5.50
  clock hsclk_p3 (source latency)               0.00       5.50
  PLL8/CKOUT3 (DUMMYPLL8)                       0.00       5.50 r
  clkmux/I3 (mx08d1)                            0.00       5.50 r
  clkmux/Z (mx08d1) (gclock source)             0.63       6.13 r
  doutregclkmux/I0 (mx02d0)                     0.00       6.13 r
  doutregclkmux/Z (mx02d0)                      0.20       6.33 r
  dout_reg/CP (dfnrb1)                          0.00       6.33 r
  dout_reg/Q (dfnrb1)                           0.32       6.65 r
  dout (out)                                    0.00       6.65 r
  data arrival time                                        6.65

  clock modeH2clkout (rise edge)                3.50       3.50
  clock hsclk_p3 (source latency)               0.00       3.50
  PLL8/CKOUT3 (DUMMYPLL8)                       0.00       3.50 f
  clkmux/I3 (mx08d1)                            0.00       3.50 f
  clkmux/Z (mx08d1) (gclock source)             0.58       4.08 f
  hsclkinv/I (inv0d0)                           0.00       4.08 f
  hsclkinv/ZN (inv0d0)                          1.00 *     5.08 r
  clkoutmux/I2 (mx04d0)                         0.00       5.08 r
  clkoutmux/Z (mx04d0)                          0.26       5.34 r
  clkout (out)                                  0.00       5.34 r
  output external delay                         0.15       5.49
  data required time                                       5.49
  ---------------------------------------------------------------
  data required time                                       5.49
  data arrival time                                       -6.65
  ---------------------------------------------------------------
  slack (MET)                                              1.15
```

Notice that the capture clock is now launched by a falling edge at the appropriate time, and goes through the hsclkinv path.

4.5 Low-speed Modes

So much for high-speed modes, now let’s look at the low-speed (divided clock) modes.

4.5.1 Mode L1

We define the settings:

```
# ModeL1 settings
set modeL1(clksel_setting) 3
set modeL1(invsel_setting) 0
set modeL1(lssel_setting) 1
```

Which implies a clock flow like this:

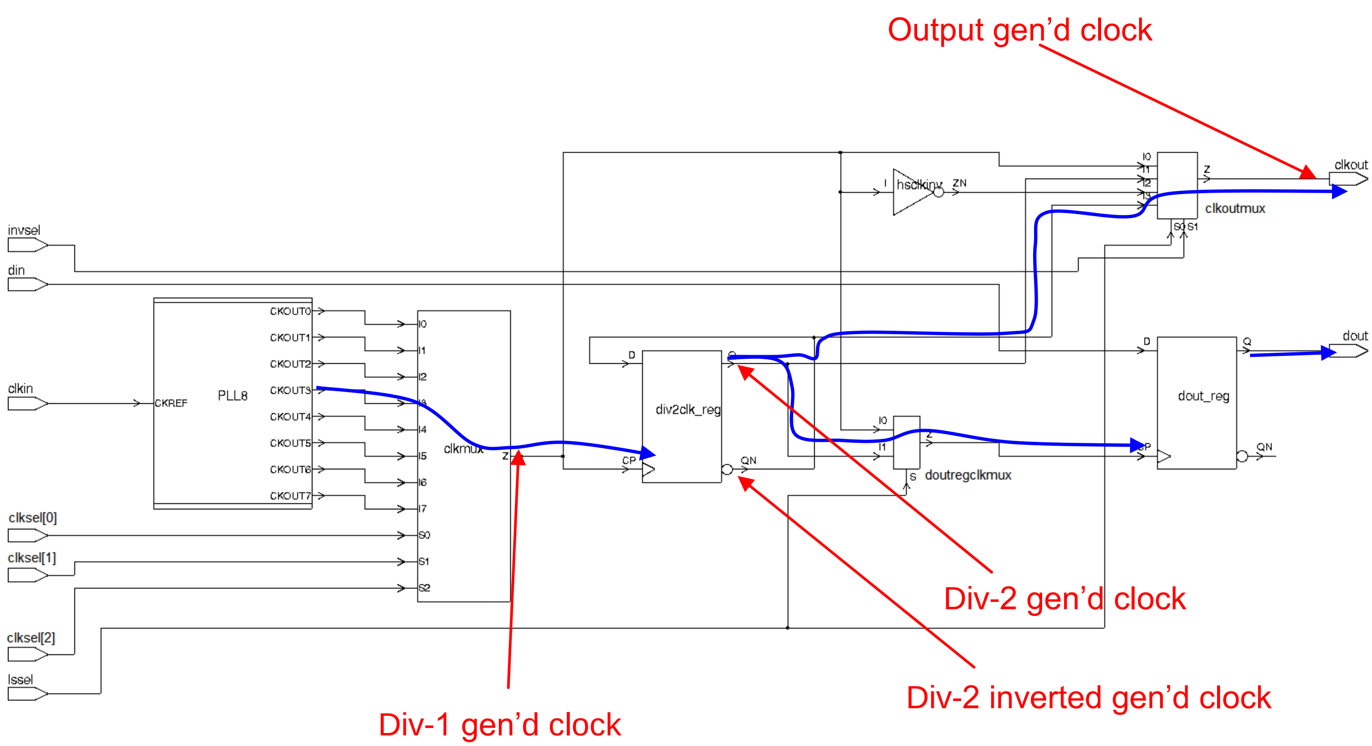


Figure 4-9

4.5.2 Creating the clocks

Create the pll tap clock based on the clksel_setting value as before:

```
# Create modeL1 clocks
create_generated_clock  \
  -name modeL1clk \
  -source [get_pins "PLL8/CKOUT${modeL1(clksel_setting)}" ] \
  -comb \
  -master_clock hsclk_p${modeL1(clksel_setting)} \
  -add \
  [get_pins clkmux/Z]
```

The dout_reg flop (in low-speed mode) always gets the divide-by 2 non-inverted clock, so we create that:

```
create_generated_clock \
  -name modeL1div2clk \
  -source [get_attribute [get_clocks modeL1clk] sources] \
  -divide_by 2 \
  -master_clock modeL1clk \
  -add \
  [get_pins div2clk_reg/Q]
```

The output clock, however, is selectable between this clock and an inverted version of this clock that comes from the QN output of the divider flop.

Since this clock is not created by an inverter, PT knows nothing about it. So, we will need to create this as a generated clock. I prefer to create it regardless of the setting of invsel_setting to make sure any noise issues are covered.

So, how do you create a divide-by 2, inverted clock.? Easy - using –divide_by 2 and –invert.

```
create_generated_clock \
  -name modeL1div2clkN \
  -source [get_attribute [get_clocks modeL1clk] sources] \
  -divide_by 2 \
  -invert \
  -master_clock modeL1clk \
  -add \
  [get_pins div2clk_reg/QN]
```

Take a look at the report_clock output (after an update_timing):

```
report_clock

Attributes:
    p - Propagated clock
    G - Generated  clock
    I - Inactive   clock

Clock            Period   Waveform            Attrs     Sources
-----------------------------------------------------------------------------
-
hsclk_p0           4.00   {0 2}               p         {PLL8/CKOUT0}
hsclk_p1           4.00   {0.5 2.5}           p         {PLL8/CKOUT1}
hsclk_p2           4.00   {1 3}               p         {PLL8/CKOUT2}
hsclk_p3           4.00   {1.5 3.5}           p         {PLL8/CKOUT3}
hsclk_p4           4.00   {2 4}               p         {PLL8/CKOUT4}
hsclk_p5           4.00   {2.5 4.5}           p         {PLL8/CKOUT5}
hsclk_p6           4.00   {3 5}               p         {PLL8/CKOUT6}
hsclk_p7           4.00   {3.5 5.5}           p         {PLL8/CKOUT7}
modeH1clk          4.00   {2.5 4.5}           p, G      {clkmux/Z}
modeH1clkout       4.00   {2.5 4.5}           p, G      {clkout}
modeH2clk          4.00   {1.5 3.5}           p, G      {clkmux/Z}
modeH2clkout       4.00   {3.5 5.5}           p, G      {clkout}
modeL1clk          4.00   {1.5 3.5}           p, G      {clkmux/Z}
modeL1div2clk      8.00   {1.5 5.5}           p, G      {div2clk_reg/Q}
modeL1div2clkN
                   8.00   {5.5 9.5}           p, G      {div2clk_reg/QN}
```

The waveform of the non-inverted divided clock is {1.5 5.5} (tap 3 means shift of 1.5). The waveform of the inverted divided clock is {5.5 9.5} (tap 3 shift of 1.5 plus half a period).

4.5.3 Handling the output clock

OK, so now we have both positive and negative phases of the divide-by 2 clock. One of these will be used as the output clock, based on the setting of invsel_setting. The simplest way to do this is just to make the master clock for clockout a variable, and use the variable in creating clkout:

```
if {$modeL1(invsel_setting) == 1} {
  set modeL1clkout_master modeL1div2clkN
} else {
  set modeL1clkout_master modeL1div2clk
}

# Finally, the output clock
create_generated_clock  \
  -name modeL1clkout \
  -source [get_attribute [get_clocks $modeL1clkout_master] sources] \
  -comb \
  -master_clock $modeL1clkout_master \
  -add \
  [get_ports clkout]
```

Set output delays and clock groups as usual:

```
set_output_delay -min [expr -1 * 0.25] -clock modeL1clkout [get_ports dout]
-add_delay
set_output_delay -max 1.7 -clock modeL1clkout [get_ports dout]  -add_delay

# Separate the modes
set_clock_groups -name muxed_out -physically_exclusive \
  -group [get_clocks modeH1*] \
  -group [get_clocks modeH2*] \
  -group [get_clocks modeL1*]
```

But this isn’t going to work. Spot the problem? Take a look at this report:

```
report_timing -delay min -to dout -path full_clock_expanded -input -group
modeL1clkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by modeL1clk)
  Endpoint: dout (output port clocked by modeL1clkout)
  Path Group: modeL1clkout
  Path Type: min

  Point                                        Incr       Path
  ---------------------------------------------------------------
  clock modeL1clk (rise edge)                  1.50       1.50
  clock hsclk_p3 (source latency)              0.00       1.50
  PLL8/CKOUT3 (DUMMYPLL8)                      0.00       1.50 r
  clkmux/I3 (mx08d1)                           0.00       1.50 r
  clkmux/Z (mx08d1) (gclock source)            0.63       2.13 r
  doutregclkmux/I0 (mx02d0)                    0.00       2.13 r
  doutregclkmux/Z (mx02d0)                     0.20       2.33 r
  dout_reg/CP (dfnrb1)                         0.00       2.33 r
  dout_reg/Q (dfnrb1)                          0.32       2.65 r
  dout (out)                                   0.00       2.65 r
  data arrival time                                       2.65
                                          Path should go through div2clk_reg!
  clock modeL1clkout (rise edge)               1.50       1.50
  clock hsclk_p3 (source latency)              0.00       1.50
  PLL8/CKOUT3 (DUMMYPLL8)                      0.00       1.50 r
  clkmux/I3 (mx08d1)                           0.00       1.50 r
  clkmux/Z (mx08d1) (gclock source)            0.63       2.13 r
  div2clk_reg/CP (dfnrb1)                      0.00       2.13 r
  div2clk_reg/Q (dfnrb1) (gclock source)
                                               0.41       2.54 r
  clkoutmux/I1 (mx04d0)                        0.00       2.54 r
  clkoutmux/Z (mx04d0)                         0.25       2.79 r
  clkout (out)                                 0.00       2.79 r
  output external delay                        0.25       3.04
  data required time                                      3.04
  ---------------------------------------------------------------
  data required time                                      3.04
  data arrival time                                      -2.65
  ---------------------------------------------------------------
  slack (VIOLATED)                                       -0.39
```

dpout_reg is getting modeL1clk directly, instead of modeL1div2clk. Because of the dourtregclkmux, the undivided modeL1clk also has a path directly to the dout_reg flop, bypassing the divider.

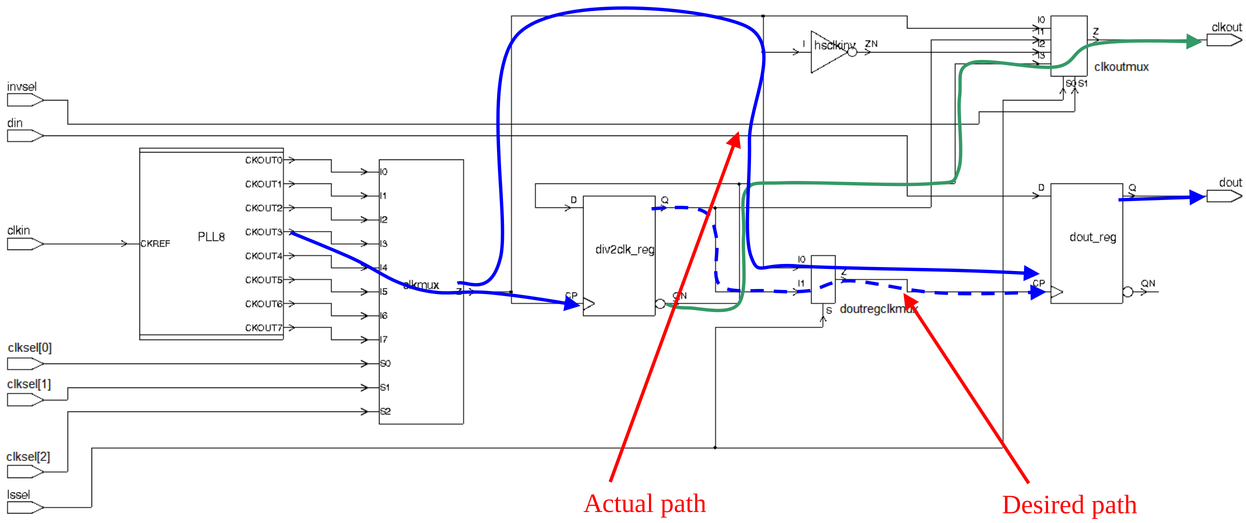


Figure 4-10

In normal operation, this path would be blocked by the programming of lssel_setting. But since we’re doing all modes in parallel, we can’t do this.

Fixing this with steering clocks could get really messy, since we don’t want to block the high-speed mode clocks.   This is one of those times when a clock kill switch is so valuable. Instead of steering clocks, we can just kill the undivided L1 clock at the mux I0 (high-speed) input:

```
set_clock_sense -stop -clock modeL1clk [get_pins doutregclkmux/I0]
```

Now the trace is correct:

```
report_timing -delay min -to dout -path full_clock_expanded -input -group
modeL1clkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by
modeL1div2clk)
  Endpoint: dout (output port clocked by modeL1clkout)
  Path Group: modeL1clkout
  Path Type: min

  Point                                        Incr       Path
  ---------------------------------------------------------------
  clock modeL1div2clk (rise edge)              1.50       1.50
  clock hsclk_p3 (source latency)              0.00       1.50
  PLL8/CKOUT3 (DUMMYPLL8)                      0.00       1.50 r
  clkmux/I3 (mx08d1)                           0.00       1.50 r
  clkmux/Z (mx08d1) (gclock source)            0.63       2.13 r
  div2clk_reg/CP (dfnrb1)                      0.00       2.13 r
  div2clk_reg/Q (dfnrb1) (gclock source)
                                               0.38       2.51 r
  doutregclkmux/I1 (mx02d0)                    0.00       2.51 r
  doutregclkmux/Z (mx02d0)                     0.17       2.68 r
  dout_reg/CP (dfnrb1)                         0.00       2.68 r
  dout_reg/Q (dfnrb1)                          0.32       3.00 r
  dout (out)                                   0.00       3.00 r
  data arrival time                                       3.00

  clock modeL1clkout (rise edge)               1.50       1.50
  clock hsclk_p3 (source latency)              0.00       1.50
  PLL8/CKOUT3 (DUMMYPLL8)                      0.00       1.50 r
  clkmux/I3 (mx08d1)                           0.00       1.50 r
  clkmux/Z (mx08d1) (gclock source)            0.63       2.13 r
  div2clk_reg/CP (dfnrb1)                      0.00       2.13 r
  div2clk_reg/Q (dfnrb1) (gclock source)
                                               0.41       2.54 r
  clkoutmux/I1 (mx04d0)                        0.00       2.54 r
  clkoutmux/Z (mx04d0)                         0.25       2.79 r
  clkout (out)                                 0.00       2.79 r
  output external delay                        0.25       3.04
  data required time                                      3.04
  ---------------------------------------------------------------
  data required time                                      3.04
  data arrival time                                      -3.00
  ---------------------------------------------------------------
  slack (VIOLATED)                                       -0.04
```

4.5.4 Mode L2

We can define another low-speed mode and use the same sort of code. But this time we'll turn invsel_setting on:

```
set modeL2(clksel_setting) 7
set modeL2(invsel_setting) 1
set modeL2(lssel_setting) 1
```

```
# Create modeL2 clocks
create_generated_clock  \
  -name modeL2clk \
  -source [get_pins "PLL8/CKOUT${modeL2(clksel_setting)}" ] \
  -comb \
  -master_clock hsclk_p${modeL2(clksel_setting)} \
  -add \
  [get_pins clkmux/Z]

# Create both pos and neg divided clocks
create_generated_clock \
  -name modeL2div2clk \
  -source [get_attribute [get_clocks modeL2clk] sources] \
  -divide_by 2 \
  -master_clock modeL2clk \
  -add \
  [get_pins div2clk_reg/Q]

create_generated_clock \
  -name modeL2div2clkN \
  -source [get_attribute [get_clocks modeL2clk] sources] \
  -divide_by 2 \
  -invert \
  -master_clock modeL2clk \
  -add \
  [get_pins div2clk_reg/QN]

if {$modeL2(invsel_setting) == 1} {
  set modeL2clkout_master modeL2div2clkN
} else {
  set modeL2clkout_master modeL2div2clk
}

# Finally, the output clock
create_generated_clock  \
  -name modeL2clkout \
  -source [get_attribute [get_clocks $modeL2clkout_master] sources] \
  -comb \
  -master_clock $modeL2clkout_master \
  -add \
  [get_ports clkout]

set_output_delay -min [expr -1 * 0.25] -clock modeL2clkout [get_ports dout]
-add_delay
set_output_delay -max 1.7 -clock modeL2clkout [get_ports dout]  -add_delay

# Separate the modes
set_clock_groups -name muxed_out -physically_exclusive \
  -group [get_clocks modeH1*] \
  -group [get_clocks modeH2*] \
  -group [get_clocks modeL1*] \
  -group [get_clocks modeL2*]
```

When doing the clock sense, instead of doing a command for each low-speed mode, we can just kill all modeL* clock coming into pin I0 of the mux:

```
set_clock_sense -stop -clock [get_clocks modeL*] [get_pins doutregclkmux/I0]
```

Similarly, we should kill any modeH* clocks going through pin I1 of the mux:

```
set_clock_sense -stop -clock [get_clocks modeH*] [get_pins doutregclkmux/I1]
```

Here's the hold timing report on dout for this mode:

```
report_timing -delay min -to dout -path full_clock_expanded -input -group
modeL2clkout

  Startpoint: dout_reg (rising edge-triggered flip-flop clocked by
modeL2div2clk)
  Endpoint: dout (output port clocked by modeL2clkout)
  Path Group: modeL2clkout
  Path Type: min

  Point                                    Incr       Path
  -------------------------------------------------------------
  clock modeL2div2clk (rise edge)         11.50      11.50
  clock hsclk_p7 (source latency)          0.00      11.50
  PLL8/CKOUT7 (DUMMYPLL8)                  0.00      11.50 r
  clkmux/I7 (mx08d1)                       0.00      11.50 r
  clkmux/Z (mx08d1) (gclock source)        0.71      12.21 r
  div2clk_reg/CP (dfnrb1)                  0.00      12.21 r
  div2clk_reg/Q (dfnrb1) (gclock source)
                                           0.38      12.60 r
  doutregclkmux/I1 (mx02d0)                0.00      12.60 r
  doutregclkmux/Z (mx02d0)                 0.17      12.76 r
  dout_reg/CP (dfnrb1)                     0.00      12.76 r
  dout_reg/Q (dfnrb1)                      0.32      13.08 r
  dout (out)                               0.00      13.08 r
  data arrival time                                  13.08

  clock modeL2clkout (rise edge)           7.50       7.50
  clock hsclk_p7 (source latency)          0.00       7.50
  PLL8/CKOUT7 (DUMMYPLL8)                  0.00       7.50 r
  clkmux/I7 (mx08d1)                       0.00       7.50 r
  clkmux/Z (mx08d1) (gclock source)        0.71       8.21 r
  div2clk_reg/CP (dfnrb1)                  0.00       8.21 r
  div2clk_reg/QN (dfnrb1) (gclock source)
                                           0.33       8.55 r
  clkoutmux/I3 (mx04d0)                    0.00       8.55 r
  clkoutmux/Z (mx04d0)                     0.25       8.80 r
  clkout (out)                             0.00       8.80 r
  output external delay                    0.25       9.05
  data required time                                  9.05
  -------------------------------------------------------------
  data required time                                  9.05
  data arrival time                                 -13.08
  -------------------------------------------------------------
  slack (MET)                                         4.03
```

4.6 Other considerations

4.6.1 H* mode divide-by 2 clocks

Although they are never used for functional purposes, the clock divider register is toggling in high-speed modes and therefore we should create the divide-by 2 clocks for proper noise analysis. This isn't strictly necessary, but it is a good habit to make sure all clocks are created, even if they aren't used as clocks in that mode.

```
create_generated_clock \
  -name modeH1div2clk \
  -source [get_attribute [get_clocks modeH1clk] sources] \
  -divide_by 2 \
  -master_clock modeH1clk \
  -add \
  [get_pins div2clk_reg/Q]

create_generated_clock \
  -name modeH1div2clkN \
  -source [get_attribute [get_clocks modeH1clk] sources] \
  -divide_by 2 \
  -invert \
  -master_clock modeH1clk \
  -add \
  [get_pins div2clk_reg/QN]

create_generated_clock \
  -name modeH2div2clk \
  -source [get_attribute [get_clocks modeH2clk] sources] \
  -divide_by 2 \
  -master_clock modeH2clk \
  -add \
  [get_pins div2clk_reg/Q]

create_generated_clock \
  -name modeH2div2clkN \
  -source [get_attribute [get_clocks modeH2clk] sources] \
  -divide_by 2 \
  -invert \
  -master_clock modeH2clk \
  -add \
  [get_pins div2clk_reg/QN]
```

The "set_clock_sense –stop_propagation" of H* through I1 of the mux shown earlier will prevent these clock from reaching the dout_reg.

4.6.2 lssel_setting

The astute reader may have noticed that we never actually used the control value lssel_setting. In a set_case_analysis flow, this value would be used to configure the circuit for either high-speed or low-speed operation. In our examples, the value was "implicit" in the mode – H1, H2, L1, L2. The creation of clocks depended on this implicit setting.

In a more generic implementation, this value might be used in a main if/then/else construct to control which set of cod is run ("H" code or "L" code)

4.6.3 Control logic

In the real circuit, it is likely that the divided and undivided clocks will go off to blocks of control logic as well as the output circuit:

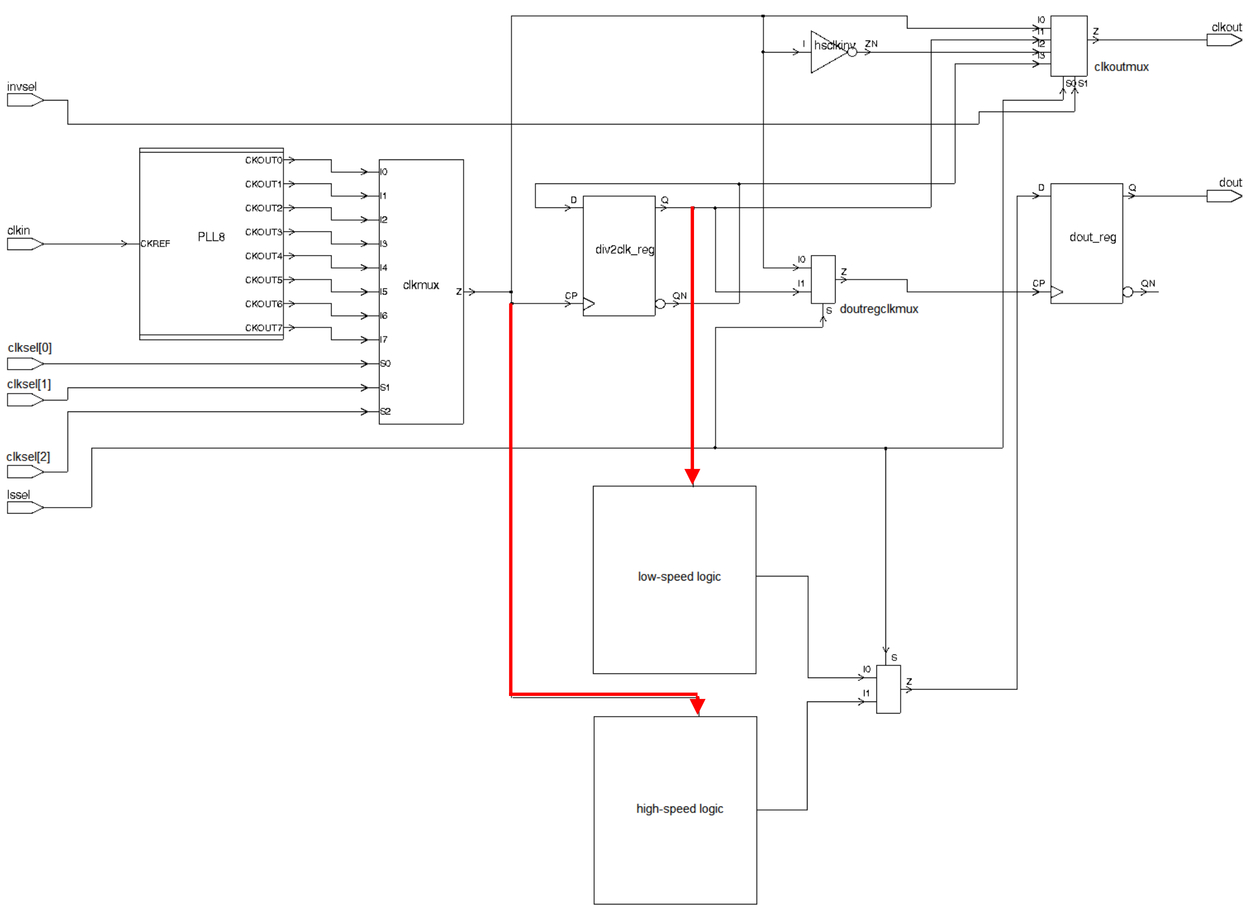


Figure 4-11

This is where the clock kill switch (set_clock_sense –stop_propagation) is absolutely essential. The logic blocks will generally be identifiable by their hierarchy paths (although the instance name itself will likely be flattened), so you can do something like:

```
foreach_in_collection clk [get_clocks modeL*] {
  set_clock_sense -stop_propagation -clock $clk [get_pins *_hsctl_*_reg/CP*]
}
```

5 Conclusion

Multiclock propagation is an essential STA technique to ensure noise-accurate analysis across all operating modes of the chip. Care must be taken to ensure that clock paths originate where they are supposed to and propagate only where they are supposed to. The latest version of PrimeTime (2006.12) provides new features that make this job much easier.

6 Acknowledgements

The author would like to acknowledge the following people for their assistance and review:

Chris Papademetrious, Synopsys
Stuart Hecht, Independent ASIC Design Consultant
Hichem Belhadj, Actel Corporation

7 References

(1) Complex Clocking Situations Using PrimeTime
Paul Zimmer
Synopsys Users Group 2001 San Jose
(available at www.zimmerdesignservices.com)

(2) Working with DDRs in PrimeTime
Paul Zimmer, Andrew Cheng
Synopsys Users Group 2002 San Jose
(available at www.zimmerdesignservices.com)

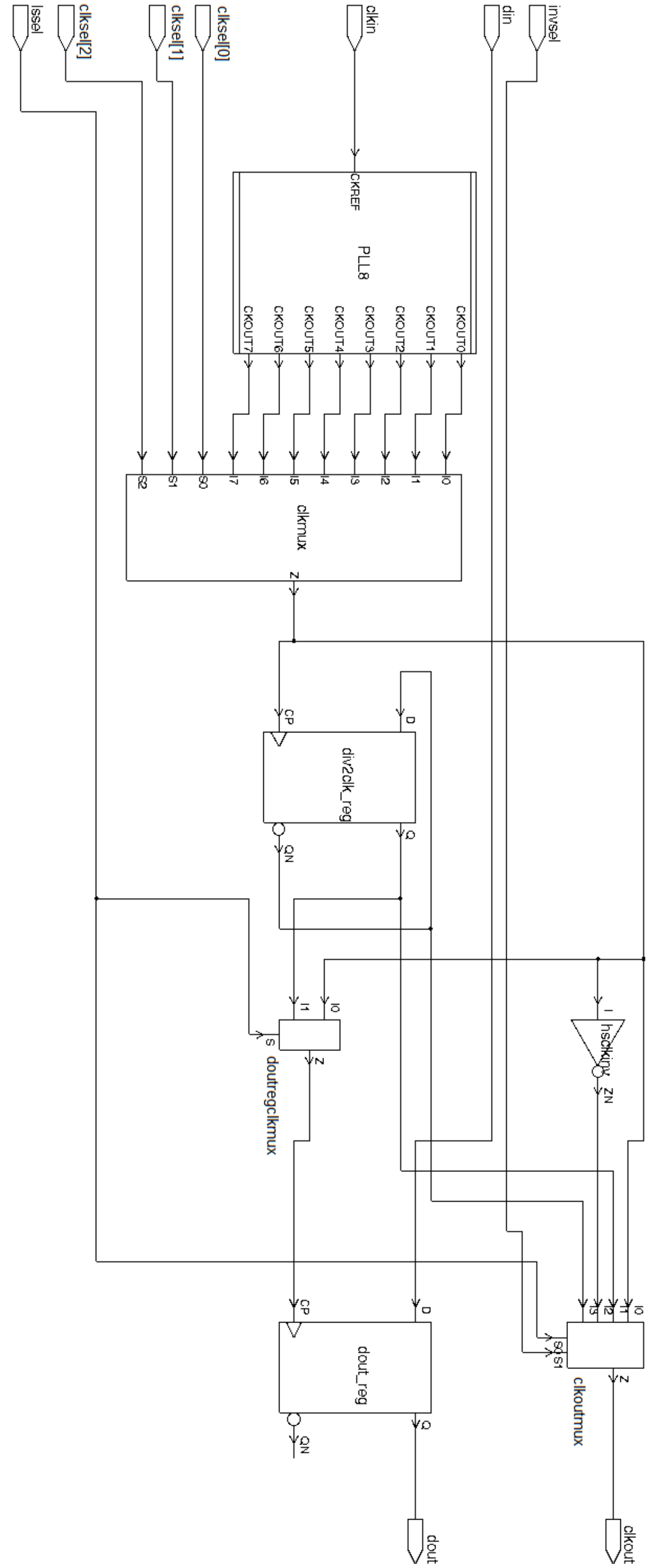
(3) My Favorite DC/PT Tcl Tricks
Paul Zimmer
Synopsys Users Group 2003 San Jose
(available at www.zimmerdesignservices.com)

(4) Working with PLLs in PrimeTime – avoiding the "phase locked oops"
Paul Zimmer
Synopsys Users Group 2005 San Jose
(available at www.zimmerdesignservices.com)

(5) Getting DDRs "write" – the 1x Output Circuit Revisited
Paul Zimmer
Synopsys Users Group 2006 San Jose
(available at www.zimmerdesignservices.com)

8 Appendix

8.1 Complex Circuit at Full Page Size


lssel
clksel[2]
clksel[1]
clksel[0]
clkin
din
invsel
CKREF
PLL8
CKOUT0
CKOUT1
CKOUT2
CKOUT3
CKOUT4
CKOUT5
CKOUT6
CKOUT7
clkmux
div2clk_reg
doutregclkmux
hsclkinv
dout_reg
clkoutmux
dout
clkout