Complex Clocking Situations Using PrimeTime

Paul Zimmer

Cisco Systems
1450 North McDowell Blvd.
Petaluma, CA 94954-6515
pzimmer@cisco.com

1.0 Introduction

Years ago, before I starting working on telecommunications chips, I used to advise other designers, "Never use the falling edge of the clock, never use divided clocks, and never mux clocks except for scan". This is still sound advice – most of these techniques should be avoided if possible. But in the telecommunications world, you just can't avoid doing these things.

In the 3 years I've spent in the telco world, I have done static timing on 5 chips, and they all had multiple edge usage, muxed clocks and divided clocks. Worse, most had circuits that did combinations of these things. Over time, I have developed some techniques for handling these situations that I thought might interest other designers. Those that do these sorts of chips for a living may already be familiar with much of what I will present, but hopefully designers who only occasionally have to deal with these sorts of problems will find some handy shortcuts here.

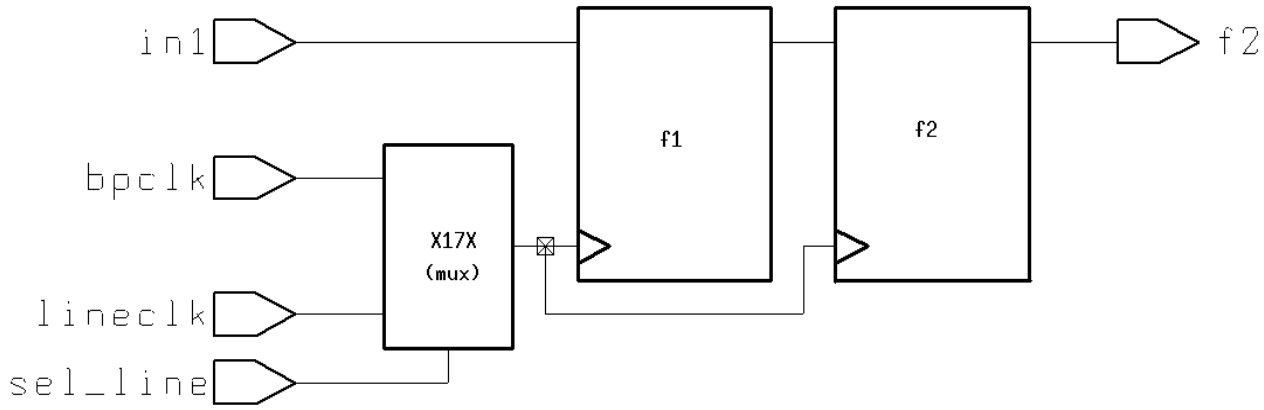
2.0 Handling Clock Muxing

When dealing with muxed clocks, it is important to do set_case_analysis on all controlling points to force the muxes into a known state. There are several reasons for this:

1) For a mux between two clocks, PT will **not** time both paths. It will time only the path involving the clock most recently created!

2) For a mux between two versions sourced by a common clock (for example, the raw clock and a delayed clock), PT will choose the worst possible scenario (data launched with the delayed clock and sampled with the raw clock for setup calculation, for example).

Here's an example. This is a simple circuit involving a muxed clock:



Read in the netlist, and do the following sequence of commands:

```
create_clock -period 10.0 [get_ports bpclk]
create_clock -period 200.0 [get_ports lineclk]
report_timing -to [get_pins f2_reg/D]
```

Since PrimeTime normally chooses the "worst case" for analysis, you would expect the timing report to use the faster clock, and check against a 10 ns period. Unfortunately, this is not the case. Instead, the result looks like this:

```
 report_timing -to [get_pins f2_reg/D] :

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by lineclk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by lineclk)
Path Group: lineclk
Path Type: max

Point                                     Incr       Path
-----------------------------------------------------------
clock lineclk (rise edge)                 0.00       0.00
clock network delay (ideal)               0.00       0.00
f1_reg/CP (FD1QA)                         0.00       0.00 r
f1_reg/Q (FD1QA)                          0.32       0.32 f
f2_reg/D (FD1QA)                          0.00       0.32 f
data arrival time                                    0.32

clock lineclk (rise edge)               200.00     200.00
clock network delay (ideal)               0.00     200.00
f2_reg/CP (FD1QA)                                  200.00 r
library setup time                       -0.29     199.71
data required time                                 199.71
-----------------------------------------------------------
data required time                                 199.71
data arrival time                                   -0.32
-----------------------------------------------------------
slack (MET)                                        199.39
```

Oops! We've got a 10ns path being checked as a 200ns path!

If we do the create_clock commands in the opposite order:

```
create_clock -period 200.0 [get_ports lineclk]
create_clock -period 10.0 [get_ports bpclk]
report_timing -to [get_pins f2_reg/D]
```

We get the 10ns check:

```
 report_timing -to [get_pins f2_reg/D] :

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by bpclk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by bpclk)
Path Group: bpclk
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock bpclk (rise edge)                  0.00       0.00
clock network delay (ideal)              0.00       0.00
f1_reg/CP (FD1QA)                        0.00       0.00 r
f1_reg/Q (FD1QA)                         0.32       0.32 f
f2_reg/D (FD1QA)                         0.00       0.32 f
data arrival time                                   0.32

clock bpclk (rise edge)                 10.00      10.00
clock network delay (ideal)              0.00      10.00
f2_reg/CP (FD1QA)                                  10.00 r
library setup time                      -0.29       9.71
data required time                                  9.71
-----------------------------------------------------------
data required time                                  9.71
data arrival time                                  -0.32
-----------------------------------------------------------
slack (MET)                                         9.39
```

But the correct way to do this is to do set_case_analysis on the mux control signal (sel_line):

```
create_clock -period 10.0 [get_ports bpclk]
create_clock -period 200.0 [get_ports lineclk]
set_case_analysis 0 [get_ports sel_line]
```

Now we get the correct calculation regardless of the order of clock declaration:

```
 report_timing -to [get_pins f2_reg/D] :

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by bpclk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by bpclk)
Path Group: bpclk
Path Type: max

Point                                     Incr       Path
-----------------------------------------------------------
clock bpclk (rise edge)                   0.00       0.00
clock network delay (ideal)               0.00       0.00
f1_reg/CP (FD1QA)                         0.00       0.00 r
f1_reg/Q (FD1QA)                          0.32       0.32 f
f2_reg/D (FD1QA)                          0.00       0.32 f
data arrival time                                    0.32

clock bpclk (rise edge)                  10.00      10.00
clock network delay (ideal)               0.00      10.00
f2_reg/CP (FD1QA)                                   10.00 r
library setup time                       -0.29       9.71
data required time                                   9.71
-----------------------------------------------------------
data required time                                   9.71
data arrival time                                   -0.32
-----------------------------------------------------------
slack (MET)                                          9.39
```

Note that situations like this can be readily found in PrimeTime using the “check_timing” command. These show up as multiple clock warnings.

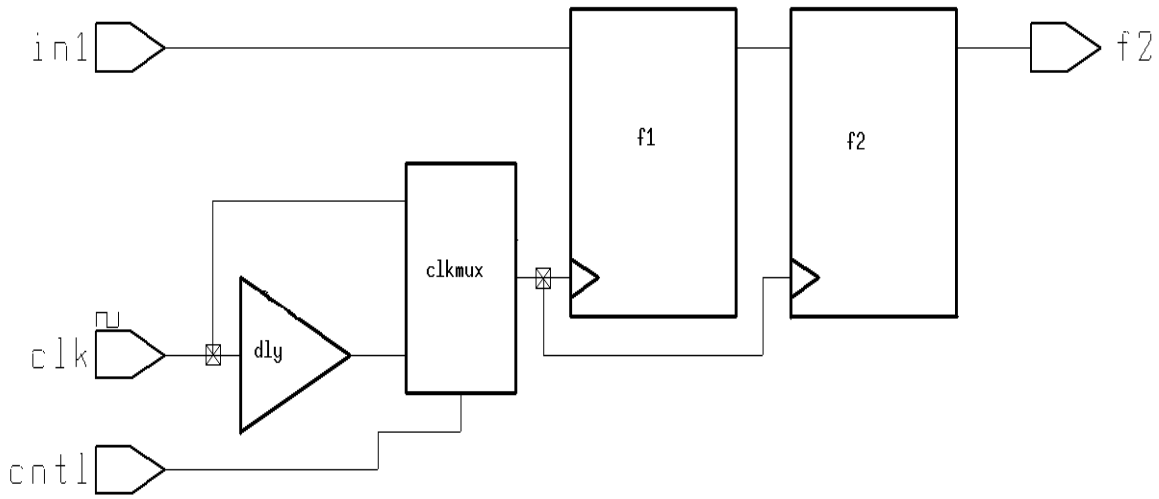
Prior to doing the set_case_analysis, check timing produced the following:

```
Warning: There are 2 clock pins which are driven by multiple clocks.
```

This warning disappears after the set_case_analysis has been done.

In general, I recommend making it part of your standard flow to clear all warnings from check_timing before proceeding with the rest of the script.

Here's an example of the other case. This is a simple circuit involving a programmable clock delay.



Read in the netlist, and do the following sequence of commands:

```
create_clock -period 10.0 [get_ports clk]
set_propagated_clock clk
set_annotated_delay 1.0 -cell -from dly/A -to dly/Z
```

(The set_annotated_delay is just to make the example easier to follow. By forcing a known delay on the dly cell, it's easier to see what's-what in the timing report)

If you report_timing to the second flop (using –path_type full_clock), you get this:

```
report_timing -to [get_pins f2_reg/D] -path_type full_clock

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by clk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Point                                     Incr       Path
-----------------------------------------------------------
clock clk (rise edge)                     0.00       0.00
clock source latency                      0.00       0.00
clk (in)                                  0.00       0.00 r
dly/Z (BUFC)                              1.00 *     1.00 r
clkmux/Z (MUX21HC)                        0.18       1.18 r
f1_reg/CP (FD1QA)                         0.00       1.18 r
f1_reg/CP (FD1QA)                         0.00       1.18 r
f1_reg/Q (FD1QA)                          0.34       1.52 f
f2_reg/D (FD1QA)                          0.00       1.52 f
data arrival time                                    1.52

clock clk (rise edge)                    10.00      10.00
clock source latency                      0.00      10.00
clk (in)                                  0.00      10.00 r
clkmux/Z (MUX21HC)                        0.20      10.20 r
f2_reg/CP (FD1QA)                         0.00      10.20 r
library setup time                       -0.27       9.93
data required time                                   9.93
-----------------------------------------------------------
data required time                                   9.93
data arrival time                                   -1.52
-----------------------------------------------------------
slack (MET)                                          8.41
```

Oops again! Look carefully at the two clock paths. PrimeTime has chosen the delayed path for the data launch, and the undelayed path for the data capture. Worse, it does the opposite for the hold check:

```
report_timing -to [get_pins f2_reg/D] -path_type full_clock -delay min

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by clk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: min

Point                                    Incr       Path
-----------------------------------------------------------
clock clk (rise edge)                    0.00       0.00
clock source latency                     0.00       0.00
clk (in)                                 0.00       0.00 r
clkmux/Z (MUX21HC)                       0.20       0.20 r
f1_reg/CP (FD1QA)                        0.00       0.20 r
f1_reg/CP (FD1QA)                        0.00       0.20 r
f1_reg/Q (FD1QA)                         0.34       0.54 f
f2_reg/D (FD1QA)                         0.00       0.54 f
data arrival time                                   0.54

clock clk (rise edge)                    0.00       0.00
clock source latency                     0.00       0.00
clk (in)                                 0.00       0.00 r
dly/Z (BUFC)                             1.00 *     1.00 r
clkmux/Z (MUX21HC)                       0.18       1.18 r
f2_reg/CP (FD1QA)                        0.00       1.18 r
f2_reg/CP (FD1QA)                                   1.18 r
library hold time                        0.16       1.34
data required time                                  1.34
-----------------------------------------------------------
data required time                                  1.34
data arrival time                                  -0.54
-----------------------------------------------------------
slack (VIOLATED)                                   -0.80
```

Again, we can use set_case_analysis to sort this mess out:

```
set_case_analysis 1 [get_ports cntl]
```

The setup and hold checks now look correct.

Here's the setup calculation:

```
report_timing -to [get_pins f2_reg/D] -path_type full_clock

 Startpoint: f1_reg (rising edge-triggered flip-flop clocked by clk)
 Endpoint: f2_reg (rising edge-triggered flip-flop clocked by clk)
 Path Group: clk
 Path Type: max

 Point                                    Incr       Path
 -----------------------------------------------------------
 clock clk (rise edge)                    0.00       0.00
 clock source latency                     0.00       0.00
 clk (in)                                 0.00       0.00 r
 dly/Z (BUFC)                             1.00 *     1.00 r
 clkmux/Z (MUX21HC)                       0.18       1.18 r
 f1_reg/CP (FD1QA)                        0.00       1.18 r
 f1_reg/CP (FD1QA)                        0.00       1.18 r
 f1_reg/Q (FD1QA)                         0.34       1.52 f
 f2_reg/D (FD1QA)                         0.00       1.52 f
 data arrival time                                   1.52

 clock clk (rise edge)                   10.00      10.00
 clock source latency                     0.00      10.00
 clk (in)                                 0.00      10.00 r
 dly/Z (BUFC)                             1.00 *    11.00 r
 clkmux/Z (MUX21HC)                       0.18      11.18 r
 f2_reg/CP (FD1QA)                        0.00      11.18 r
 library setup time                      -0.27      10.91
 data required time                                 10.91
 -----------------------------------------------------------
 data required time                                 10.91
 data arrival time                                  -1.52
 -----------------------------------------------------------
 slack (MET)                                         9.39
```

And here's the hold calculation:

```
report_timing -to [get_pins f2_reg/D] -path_type full_clock

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by clk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: min

Point                                    Incr       Path
-----------------------------------------------------------
clock clk (rise edge)                    0.00       0.00
clock source latency                     0.00       0.00
clk (in)                                 0.00       0.00 r
dly/Z (BUFC)                             1.00 *     1.00 r
clkmux/Z (MUX21HC)                       0.18       1.18 r
f1_reg/CP (FD1QA)                        0.00       1.18 r
f1_reg/CP (FD1QA)                        0.00       1.18 r
f1_reg/Q (FD1QA)                         0.34       1.52 f
f2_reg/D (FD1QA)                         0.00       1.52 f
data arrival time                                   1.52

clock clk (rise edge)                    0.00       0.00
clock source latency                     0.00       0.00
clk (in)                                 0.00       0.00 r
dly/Z (BUFC)                             1.00 *     1.00 r
clkmux/Z (MUX21HC)                       0.18       1.18 r
f2_reg/CP (FD1QA)                        0.00       1.18 r
f2_reg/CP (FD1QA)                                   1.18 r
library hold time                        0.16       1.34
data required time                                  1.34
-----------------------------------------------------------
data required time                                  1.34
data arrival time                                  -1.52
-----------------------------------------------------------
slack (MET)                                         0.18
```

If we had done the set_case_analysis to 0 instead of one, the slack numbers would have been the same, since the "1.00" gets added on in both parts of the calculation. Here's the hold report again with the set_case_analysis set to 0:

```
report_timing -to [get_pins f2_reg/D] -path_type full_clock -delay min

Startpoint: f1_reg (rising edge-triggered flip-flop clocked by clk)
Endpoint: f2_reg (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: min

Point                                          Incr       Path
-----------------------------------------------------------------
clock clk (rise edge)                          0.00       0.00
clock source latency                           0.00       0.00
clk (in)                                       0.00       0.00 r
clkmux/Z (MUX21HC)                             0.20       0.20 r
f1_reg/CP (FD1QA)                              0.00       0.20 r
f1_reg/CP (FD1QA)                              0.00       0.20 r
f1_reg/Q (FD1QA)                               0.34       0.54 f
f2_reg/D (FD1QA)                               0.00       0.54 f
data arrival time                                         0.54

clock clk (rise edge)                          0.00       0.00
clock source latency                           0.00       0.00
clk (in)                                       0.00       0.00 r
clkmux/Z (MUX21HC)                             0.20       0.20 r
f2_reg/CP (FD1QA)                              0.00       0.20 r
f2_reg/CP (FD1QA)                                         0.20 r
library hold time                              0.16       0.36
data required time                                        0.36
-----------------------------------------------------------------
data required time                                        0.36
data arrival time                                        -0.54
-----------------------------------------------------------------
slack (MET)                                               0.18
```

Simple enough on these toy examples. But in the real world these sorts of mux selects are usually controlled by registers, often by processor configuration registers. Furthermore, it's likely to be a bitfield in a larger register. That's a pain to do. What we need is a little PrimeTime proc that can be used to do set_case_analysis of a binary value to a bit range in a grouped set of registers.

There may be more clever ways to do this in tcl, but here's my version:

```
# _set_ctl_reg - put set_case_analysis values on an ordered group of pins
# syntax:
# _set_ctl_reg register_name binary_value [start_bit]
#
proc _set_ctl_reg {_reg_name _binary_value {_start_bit 0}} {
  global _bus_bit_divider
  global _bus_bit_divider_open
  global _bus_bit_divider_close
  global _Qpin_name
  global _QNpin_name

  # Default variables
  if {[info exists _bus_bit_divider]} {
    set _bus_bit_divider_open $_bus_bit_divider
    set _bus_bit_divider_close $_bus_bit_divider
  }
  if !{[info exists _bus_bit_divider_open]} {set _bus_bit_divider_open _}
  if !{[info exists _bus_bit_divider_close]} {set _bus_bit_divider_close _}
  if !{[info exists _Qpin_name]} { set _Qpin_name "Q" }
  if !{[info exists _QNpin_name]} { set _QNpin_name "QN" }

  set _length [string length $_binary_value]
  set _last_bit [expr $_length + $_start_bit - 1]

  set _bit $_start_bit
  set _string_index [expr $_length - 1]
  while {$_bit <= $_last_bit} {
    # get target pin value
    set _value [string index $_binary_value $_string_index]
    # set inverted value for QN
    if {$_value == 1} {
      set _notvalue 0
    } else {
      set _notvalue 1
    }
    echo "-> Setting bit $_bit in $_reg_name to $_value"
    set _pin [get_pins
${_reg_name}${_bus_bit_divider_open}${_bit}${_bus_bit_divider_close}*/$_Qpin_n
ame]
    echo "  -> Q pin [get_object_name $_pin]"
    set_case_analysis $_value $_pin
    set _pin [get_pins -quiet
${_reg_name}${_bus_bit_divider_open}${_bit}${_bus_bit_divider_close}*/$_QNpin_
name]
    if {[sizeof_collection $_pin] != 0} {
      echo "  -> QN pin [get_object_name $_pin]"
      set_case_analysis $_notvalue $_pin
    }
    set _bit [expr $_bit + 1]
    set _string_index [expr $_string_index - 1]
  }
}
```

It is important to note that the proc needs to do the set_case_analysis on *both* the Q and QN pins, since you can't be sure which might be used.

The global variables can be set externally to override the defaults for the bus bit divider character(s) and the Q and QN pin names.

To set the value "1111" on the bits of Acore/upif/muxctl_reg_5_ down to Acore/upif/muxctl_reg_2_, you'd do this:

```
_set_ctl_reg  Acore/upif/muxctl_reg  1111 2
```

3.0 Managing large numbers of clocks

Most designs these days have multiple clocks. It is important to properly manage the paths between these clock domains.

If the clocks are completely asynchronous to one another, I usually do set_false_path between the clocks like this:

```
set_false_path -from [get_clocks clk1] -to [get_clocks clk2]
set_false_path -from [get_clocks clk2] -to [get_clocks clk1]
```

Some people may disagree with this approach. It will mask *all* paths between the clocks. My view is that, if the clocks are truly asynchronous, it is the designers' responsibility to handle these paths. The static timing tool can only point them out, it cannot really time them. And on a large chip, just pointing them out can be an immense task. So, I false_path them.

Sometimes, however, the clocks are what I call "related clocks", meaning they are derived from a common source and thus have fixed phase relationships. The most obvious example would be a master clock and a divide-by-2 clock derived from it. Provided that you do *not* do set_false_path between these clocks, PrimeTime will calculate the phase difference according to the actual gate and interconnect delays, and will therefore time the circuit correctly.

With a small number of clocks, this can be readily managed by doing pairs of set_false_path statements between unrelated clocks as above. Unfortunately, as the number of clocks goes up this quickly becomes unmanageable and unreadable.

The telco designs I work with commonly have dozens or even hundreds of clocks. That's why I have developed some processes and scripting techniques to automate all of this. The basic idea is fairly simple – create an array to map generated (related) clocks to their master sources, and use this information to do the false paths.

Here's the basic flow:

1. Keep track of the mapping of clocks to their parents in an array (called _gen2src) as the generated clocks are created. Ideally, it would be nice to just hang this on the clock as an attribute. But PrimeTime won't allow you to do this (at least, it wouldn't the last time I tried), so I store it in an array instead.

Note that it is possible to have a generated clock generated from another generated clock. It has been my experience that this only works correctly if the "–source" argument refers to the original, non-generated source clock *source pin*. Similarly, the entry in the _gen2src array should refer to this original master clock, but by clock name rather than by source pin.

2. Loop through all the clocks and do set_false_path pairwise between them. It might also work to do a single pair of set_false path with [all_clocks] as both the -to and -from, but I like to see the pair-wise output.

3. Go back and remove the false path between the related clocks using the _gen2src array.

Here's sample code to do this:

```
# Create the clocks
set _pll_clk311_src [get_pins pllmn/ICLK]
create_clock -period $_311_period -name pll_clk311 $_pll_clk311_src
set _clkname ft_plldiv2_clk
create_generated_clock \
    -name $_clkname \
    -source $_pll_clk311_src \
    -divide_by 2 \
    [get_pins Acore/freq/counter311_reg_0_/Q]
  set _gen2src($_clkname) "pll_clk311"

# Set false paths pair-wise
set _all_clks [all_clocks]  # Should work with ptime >= 9910
foreach_in_collection _clk $_all_clks {
  foreach_in_collection _other_clk [remove_from_collection $_all_clks $_clk] {
    set_false_path -from $_clk -to $_other_clk
    # keep track in assoc arrays
    set _sfp_clks([get_object_name $_clk],[get_object_name $_other_clk]) 1
  }
}

# Remove the false_path between related clocks
foreach_in_collection _clk1 $_all_clks {
  # Initialize a collection to contain the clock itself
  set _clk1_clks $_clk1
  # set up name
  set _clk1_name [get_object_name $_clk1]
  # Now go through gen2src and add any related generated clocks to the
  # collection
  foreach _gen_clk_name [array names _gen2src] {
    if {$_gen2src($_gen_clk_name) == $_clk1_name} {
      set _clk1_clks [add_to_collection $_clk1_clks [get_clock
$_gen_clk_name]]
    }
  }

  # Now reset the paths.  Do this pair by pair to get the comments.
  foreach_in_collection _clk1_clk $_clk1_clks {
    foreach_in_collection _other_clk1_clk \
        [remove_from_collection $_clk1_clks $_clk1_clk] {
      reset_path -from $_clk1_clk -to $_other_clk1_clk
      reset_path -from $_other_clk1_clk -to $_clk1_clk
      # keep track in assoc arrays
      set _sfp_clks([get_object_name $_clk1_clk],[get_object_name
$_other_clk1_clk]) 0
      set _sfp_clks([get_object_name $_other_clk1_clk],[get_object_name
$_clk1_clk]) 0
    }
  }
}
```

Notes:

1. The routines use a variable called "_all_clks" that is expected to contain all the clocks (as a collection). In recent version of PrimeTime, this can be safely done with "[all_clocks]". In older versions, the generated clocks might or might not be returned, depending on whether a timing update had been done recently.

2. Generated clocks have both a "clock" object and a "generated clock" object associated with them. It has been my experience that things work better if you always do set_false_path, set_multicycle_path, etc on the *clock* object.

3. The script builds an associative array "_sfp_clks" which can later be interrogated to see what pairs actually have false path set between them.

Here's a PrimeTime script to dump the contents of _sfp_clks:

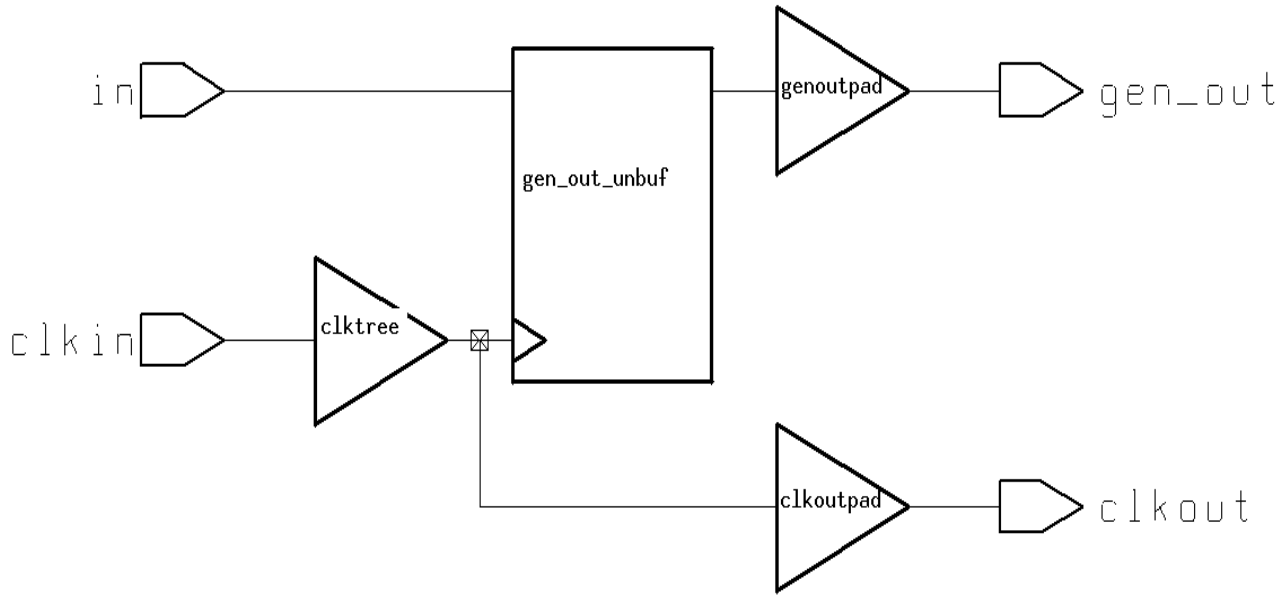
```
foreach _key [lsort -ascii [array names "_sfp_clks"]] {
    # break the key into from/to pieces
    set _from_to [split $_key ,]
    set _from [lindex $_from_to 0]
    set _to [lindex $_from_to 1]

    # handle the disabled paths
    if {$_sfp_clks($_key) == 0} {
      if {[info exists _sfp_clks($_to,$_from)]} {
        if {$_sfp_clks($_to,$_from) == 0} {
          echo "Clock path enabled between $_from and $_to"
          if !{[info exists _seen($_to)]} {
            set _seen($_from) 1
          }
        } else {
          echo -n "-> Err"; echo "or: Clock path enabled from $_from to $_to,
but not the reverse!"
        }
      } else {
          echo -n "-> Err"; echo "or: _sfp_clks has an entry \"$_from,$_to\",
but no entry \"$_to,$_from\""
      }
    }
  }
```

4.0 I/O interfaces with outgoing clocks

Most modern I/O interfaces send the clock along with the data. This avoids the problem of having to make a round-trip through the chip in a single clock cycle that was a common characteristic of older, externally clocked outputs, and allows the interface speed to be limited only by skew rather than absolute delays.

4.1 Example 1: Basic outgoing clock

In this circuit, a clock comes in, goes through a clock distribution network, and clocks a flop. The flop output and the clock are then sent off chip. Buffers are used to represent the clock tree and pads:



With PrimeTime versions prior to 1999.05, this was a very nasty circuit to time. The basic technique was to use get_timing_paths to time the outgoing clock path, then create an external virtual clock with this timing and do set_output_delay relative to this virtual clock. Simple enough in principle, but very nasty when you start doing more complicated things like divided clocks and opposite edges.

Fortunately, PrimeTime now allows this to be done in a much more straightforward manner by creating a generated clock with a divide-by of ***1***.

```
create_clock -period 10.0 [get_ports clkin]
set_propagated_clock clkin
create_generated_clock \
    -name clkout \
    -source [get_ports clkin] \
    -divide_by 1 \
    [get_ports clkout]
set _gen2src(clkout) clkin

set_output_delay -clock clkout 1.0 gen_out
```

Now create a difference in output timing:

```
set_load 0.5 [get_ports gen_out]
set_load 0.25 [get_ports clkout]
```

```
set_annotated_delay -cell -from clkoutpad/A -to clkoutpad/Z -rise 0.3
set_annotated_delay -cell -from clkoutpad/A -to clkoutpad/Z -fall 0.1
```

And this is the resulting timing report:

```
  report_timing -input_pins -path_type full_clock -to gen_out

  Startpoint: gen_out_unbuf_reg
              (rising edge-triggered flip-flop clocked by clkin)
  Endpoint: gen_out (output port clocked by clkout)
  Path Group: clkout
  Path Type: max

  Point                                     Incr       Path
  -----------------------------------------------------------
  clock clkin (rise edge)                   0.00       0.00
  clock source latency                      0.00       0.00
  clkin (in)                                0.00       0.00 r
  clktree/A (BUFC)                          0.00       0.00 r
  clktree/Z (BUFC)                          0.11       0.11 r
  gen_out_unbuf_reg/CP (FD1QA)              0.00       0.11 r
  gen_out_unbuf_reg/CP (FD1QA)              0.00       0.11 r
  gen_out_unbuf_reg/Q (FD1QA)               0.33       0.44 r
  genoutpad/A (BUFC)                        0.00       0.44 r
  genoutpad/Z (BUFC)                        0.42       0.86 r
  gen_out (out)                             0.00       0.86 r
  data arrival time                                    0.86

  clock clkout (rise edge)                 10.00      10.00
  clock network delay (ideal)               0.41      10.41
  output external delay                    -1.00       9.41
  data required time                                   9.41
  -----------------------------------------------------------
  data required time                                   9.41
  data arrival time                                   -0.86
  -----------------------------------------------------------
  slack (MET)                                          8.55
```

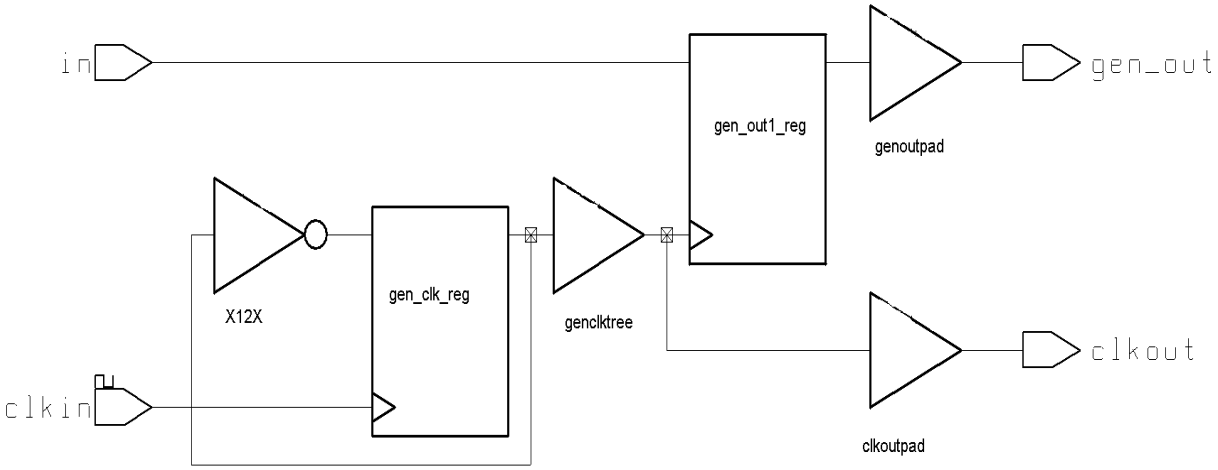
Note the “0.41” as the clock network delay of clkout. Here’s where the number comes from:

```
 report_timing -delay max_rise -to [get_ports clkout]

Startpoint: clkin (clock source 'clkin')
Endpoint: clkout (output port)
Path Group: (none)
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clkin (in)                               0.00       0.00 r
clktree/Z (BUFC)                         0.11       0.11 r
clkoutpad/Z (BUFB)                       0.30 *     0.41 r
clkout (out)                             0.00       0.41 r
data arrival time                                   0.41
```

It is also interesting to note that this clock network delay appears even though I didn’t do set_propagated_clock on clkout. I believe that is because PrimeTime treats it like a kind of source latency.

4.2 Example 2: Divided outgoing clock

Now let’s look at a more complicated example. The following circuit generates a divide-by 2 clock, uses this to clock the outgoing data flop, and sends the divided clock out as well:



We read in the netlist and do:

```
create_clock -period 10.0 [get_ports clkin]
set_propagated_clock clkin

# create the div2 clock on the port
create_generated_clock \
    -name clkout \
    -source [get_ports clkin] \
    -divide_by 2 \
    [get_ports clkout]

# create the div2 clock on the Q pin (so that it gets used for gen_out flop)
create_generated_clock \
    -name clkout_at_q \
    -source [get_ports clkin] \
    -divide_by 2 \
    [get_pins gen_clk_reg/Q]
set_propagated_clock clkout_at_q

set_output_delay -clock clkout 1.0 gen_out
```

A couple of things to note about the above script:

1. The create_generated_clock command is used twice – once to create the divide-by-2 clock at the divider flop, and again to create the outgoing clock. The first is necessary because the outgoing data flop uses this clock; the second is necessary so that we can tie the output port to the real outgoing clock.

2. Both the main clock and the divide-by-2 clock at the divider flop are set as propagated. This is necessary to get the correct timing. Setting propagated_clock on the outgoing clock is not necessary, and I have found it a good idea to avoid setting propagated clock on anything (such as a virtual clock) that doesn't really have anything to propagate through.

To make the timing report easier to follow, we'll again add some loads and delays:

```
set_load 0.5 [get_ports gen_out]
set_load 0.25 [get_ports clkout]
```

And here is the resulting timing trace:

```
report_timing -to [get_ports gen_out] -input_pins -path_type full_clock

 Startpoint: gen_out1_reg
             (rising edge-triggered flip-flop clocked by clkout_at_q)
 Endpoint: gen_out (output port clocked by clkout)
 Path Group: clkout
 Path Type: max

 Point                                    Incr       Path
 -----------------------------------------------------------
 clock clkout_at_q (rise edge)            0.00       0.00
 clock source latency                     0.34       0.34
 gen_clk_reg/Q (FD1QA)                    0.00       0.34 r
 genclktree/A (BUFC)                      0.00       0.34 r
 genclktree/Z (BUFC)                      0.14       0.48 r
 gen_out1_reg/CP (FD1QA)                  0.00       0.48 r
 gen_out1_reg/CP (FD1QA)                  0.00       0.48 r
 gen_out1_reg/Q (FD1QA)                   0.33       0.82 r
 genoutpad/A (BUFC)                       0.00       0.82 r
 genoutpad/Z (BUFC)                       0.42       1.24 r
 gen_out (out)                            0.00       1.24 r
 data arrival time                                   1.24

 clock clkout (rise edge)                20.00      20.00
 clock network delay (ideal)              0.81      20.81
 output external delay                   -1.00      19.81
 data required time                                  19.81
 -----------------------------------------------------------
 data required time                                  19.81
 data arrival time                                   -1.24
 -----------------------------------------------------------
 slack (MET)                                         18.57
```

Let's track the various pieces.

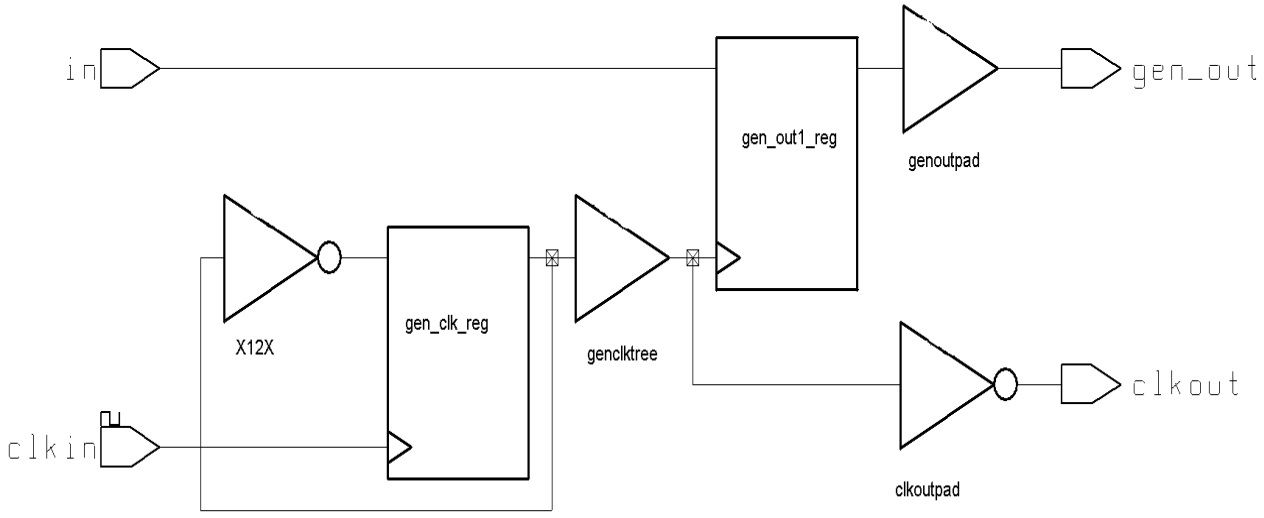
First, here's where the "clock source latency" of clkout_at_q comes from:

```
   report_timing -from gen_clk_reg/CP -to gen_clk_reg/Q

Startpoint: gen_clk_reg
            (rising edge-triggered flip-flop clocked by clkin)
Endpoint: gen_clk_reg/Q (internal pin)
Path Group: (none)
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
gen_clk_reg/CP (FD1QA)                   0.00       0.00 r
gen_clk_reg/Q (FD1QA)                    0.34       0.34 r
data arrival time                                   0.34
```

The "clock network delay (ideal)" of clkout is this value plus the delay through the genclktree and the clkout pad:

```
report_timing -to [get_ports clkout]

Startpoint: gen_clk_reg/Q
            (clock source 'clkout_at_q')
Endpoint: clkout (output port)
Path Group: (none)
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
clock source latency                     0.34       0.34
gen_clk_reg/Q (FD1QA)                    0.00       0.34 r
genclktree/Z (BUFC)                      0.14       0.48 r
clkoutpad/Z (BUFB)                       0.33       0.81 r
clkout (out)                             0.00       0.81 r
data arrival time                                   0.81
```

Now, suppose you've got a lot of clock period to work with, and you decide to send the clock out inverted to avoid the hold time problems common to this sort of "clock + data" interface. Now the circuit looks like this (note that clkoutpad is now an inverter):



It would be nice if PrimeTime recognized the inversion and handled this automatically. It does not. You have to tell it about the inversion using the "-invert" switch on create_generated_clock. Here is the script now:

```
create_clock -period 10.0 [get_ports clkin]
set_propagated_clock clkin

# create the div2 clock on the port
create_generated_clock \
    -name clkout \
    -source [get_ports clkin] \
    -divide_by 2 \
    -invert \
    [get_ports clkout]

# create the div2 clock on the Q pin (so that it gets used for gen_out flop)
create_generated_clock \
    -name clkout_at_q \
    -source [get_ports clkin] \
    -divide_by 2 \
    [get_pins gen_clk_reg/Q]
set_propagated_clock clkout_at_q

set_output_delay -clock clkout 1.0 gen_out

# create a difference in output timing
set_load 0.5 [get_ports gen_out]
set_load 0.25 [get_ports clkout]
```

And here's the resulting timing trace:

```
  report_timing -to [get_ports gen_out] -input_pins -path_type full_clock

Startpoint: gen_out1_reg
            (rising edge-triggered flip-flop clocked by clkout_at_q)
Endpoint: gen_out (output port clocked by clkout)
Path Group: clkout
Path Type: max

Point                                     Incr       Path
-------------------------------------------------------------
clock clkout_at_q (rise edge)             0.00       0.00
clock source latency                      0.34       0.34
gen_clk_reg/Q (FD1QA)                     0.00       0.34 r
genclktree/A (BUFC)                       0.00       0.34 r
genclktree/Z (BUFC)                       0.15       0.49 r
gen_out1_reg/CP (FD1QA)                   0.00       0.49 r
gen_out1_reg/CP (FD1QA)                   0.00       0.49 r
gen_out1_reg/Q (FD1QA)                    0.34       0.83 r
genoutpad/A (BUFC)                        0.00       0.83 r
genoutpad/Z (BUFC)                        0.42       1.25 r
gen_out (out)                             0.00       1.25 r
data arrival time                                    1.25

clock clkout (rise edge)                 10.00      10.00
clock network delay (ideal)               0.77      10.77
output external delay                    -1.00       9.77
data required time                                   9.77
-------------------------------------------------------------
data required time                                   9.77
data arrival time                                   -1.25
-------------------------------------------------------------
slack (MET)                                          8.52
```

Notice that we're still launching data at time 0, but we're now checking to at time 10.0. This is correct, because the inverted output clock only allows half a cycle for setup - the hold calculation will get the other 10.0 ns:

```
  report_timing -to [get_ports gen_out] -input_pins -path_type full_clock -
delay min

  Startpoint: gen_out1_reg
              (rising edge-triggered flip-flop clocked by clkout_at_q)
  Endpoint: gen_out (output port clocked by clkout)
  Path Group: clkout
  Path Type: min

  Point                                     Incr       Path
  ------------------------------------------------------------
  clock clkout_at_q (rise edge)           20.00      20.00
  clock source latency                     0.34      20.34
  gen_clk_reg/Q (FD1QA)                    0.00      20.34 r
  genclktree/A (BUFC)                      0.00      20.34 r
  genclktree/Z (BUFC)                      0.15      20.49 r
  gen_out1_reg/CP (FD1QA)                  0.00      20.49 r
  gen_out1_reg/CP (FD1QA)                  0.00      20.49 r
  gen_out1_reg/Q (FD1QA)                   0.34      20.83 f
  genoutpad/A (BUFC)                       0.00      20.83 f
  genoutpad/Z (BUFC)                       0.36      21.19 f
  gen_out (out)                            0.00      21.19 f
  data arrival time                                  21.19

  clock clkout (rise edge)                10.00      10.00
  clock network delay (ideal)              0.77      10.77
  output external delay                   -1.00       9.77
  data required time                                  9.77
  ------------------------------------------------------------
  data required time                                  9.77
  data arrival time                                 -21.19
  ------------------------------------------------------------
  slack (MET)                                        11.42
```

To know this is correct, we need to see where that "0.77" clock network delay on clkout comes from:

```
report_timing -to [get_ports clkout] -delay max_rise

 Startpoint: gen_clk_reg/Q
             (clock source 'clkout_at_q')
 Endpoint: clkout (output port)
 Path Group: (none)
 Path Type: max

 Point                                    Incr       Path
 -----------------------------------------------------------
 clock source latency                     0.34       0.34
 gen_clk_reg/Q (FD1QA)                    0.00       0.34 f
 genclktree/Z (BUFC)                      0.17       0.50 f
 clkoutpad/Z (N1B)                        0.27       0.77 r
 clkout (out)                             0.00       0.77 r
 data arrival time                                   0.77
```

Note that I did the report_timing with "-delay max_rise" because the *external* device is using the rising edge of the clock. Notice that PrimeTime was smart enough to follow the inversion in the path and track this back to a falling edge of gen_clk_reg/Q.

Now, suppose the external device were using the *falling* edge, and we still wanted to do opposite edge clocking. We can do this by going back to the original circuit. But the script now needs to tell PrimeTime that the device captures data on falling edges. The script looks the same as the original script, with one change. The set_output_delay command now needs the "-clock_fall" switch:

```
set_output_delay -clock clkout 1.0 gen_out -clock_fall
```

Here's the resulting timing trace:

```
report_timing -to [get_ports gen_out] -input_pins -path_type full_clock


Startpoint: gen_out1_reg
            (rising edge-triggered flip-flop clocked by clkout_at_q)
Endpoint: gen_out (output port clocked by clkout)
Path Group: clkout
Path Type: max

Point                                       Incr       Path
-------------------------------------------------------------
clock clkout_at_q (rise edge)               0.00       0.00
clock source latency                        0.34       0.34
gen_clk_reg/Q (FD1QA)                       0.00       0.34 r
genclktree/A (BUFC)                         0.00       0.34 r
genclktree/Z (BUFC)                         0.14       0.48 r
gen_out1_reg/CP (FD1QA)                     0.00       0.48 r
gen_out1_reg/CP (FD1QA)                     0.00       0.48 r
gen_out1_reg/Q (FD1QA)                      0.33       0.82 r
genoutpad/A (BUFC)                          0.00       0.82 r
genoutpad/Z (BUFC)                          0.42       1.24 r
gen_out (out)                               0.00       1.24 r
data arrival time                                      1.24

clock clkout (fall edge)                   10.00      10.00
clock network delay (ideal)                 0.78      10.78
output external delay                      -1.00       9.78
data required time                                     9.78
-------------------------------------------------------------
data required time                                     9.78
data arrival time                                     -1.24
-------------------------------------------------------------
slack (MET)                                            8.54
```

Again, we have to show where the "0.78" clock network delay comes from. Since the external device uses the falling edge, we need to use the "-delay max_fall" switch:

```
report_timing -to [get_ports clkout] -delay max_fall

Startpoint: gen_clk_reg/Q
            (clock source 'clkout_at_q')
Endpoint: clkout (output port)
Path Group: (none)
Path Type: max

Point                                       Incr       Path
-------------------------------------------------------------
clock source latency                        0.34       0.34
gen_clk_reg/Q (FD1QA)                       0.00       0.34 f
genclktree/Z (BUFC)                         0.16       0.50 f
clkoutpad/Z (BUFB)                          0.28       0.78 f
clkout (out)                                0.00       0.78 f
data arrival time                                      0.78
```

And, yes, it does work when you do both. This corresponds to an external device that clocks on the falling edge and a circuit that sends out an inverted clock. This means that the data will transition on the same edge that it is sampled, so we expect the setup check to use the full 20ns of the divided clock. And it does. Here's the setup timing trace when the "-invert" switch is used on the create_generated_clock AND the "-clock_fall" switch is used on the set_output_delay:

```
report_timing -to [get_ports gen_out] -input_pins -path_type full_clock

Startpoint: gen_out1_reg
            (rising edge-triggered flip-flop clocked by clkout_at_q)
Endpoint: gen_out (output port clocked by clkout)
Path Group: clkout
Path Type: max

Point                                       Incr       Path
-----------------------------------------------------------
clock clkout_at_q (rise edge)               0.00       0.00
clock source latency                        0.34       0.34
gen_clk_reg/Q (FD1QA)                       0.00       0.34 r
genclktree/A (BUFC)                         0.00       0.34 r
genclktree/Z (BUFC)                         0.15       0.49 r
gen_out1_reg/CP (FD1QA)                     0.00       0.49 r
gen_out1_reg/CP (FD1QA)                     0.00       0.49 r
gen_out1_reg/Q (FD1QA)                      0.34       0.83 r
genoutpad/A (BUFC)                          0.00       0.83 r
genoutpad/Z (BUFC)                          0.42       1.25 r
gen_out (out)                               0.00       1.25 r
data arrival time                                      1.25

clock clkout (fall edge)                   20.00      20.00
clock network delay (ideal)                 0.68      20.68
output external delay                      -1.00      19.68
data required time                                    19.68
-----------------------------------------------------------
data required time                                    19.68
data arrival time                                     -1.25
-----------------------------------------------------------
slack (MET)                                           18.43
```

And here's the corresponding hold trace:

```
    report_timing -to [get_ports gen_out] -input_pins -path_type full_clock -
delay min

  Startpoint: gen_out1_reg
              (rising edge-triggered flip-flop clocked by clkout_at_q)
  Endpoint: gen_out (output port clocked by clkout)
  Path Group: clkout
  Path Type: min

  Point                                       Incr       Path
  ------------------------------------------------------------
  clock clkout_at_q (rise edge)               0.00       0.00
  clock source latency                        0.34       0.34
  gen_clk_reg/Q (FD1QA)                       0.00       0.34 r
  genclktree/A (BUFC)                         0.00       0.34 r
  genclktree/Z (BUFC)                         0.15       0.49 r
  gen_out1_reg/CP (FD1QA)                     0.00       0.49 r
  gen_out1_reg/CP (FD1QA)                     0.00       0.49 r
  gen_out1_reg/Q (FD1QA)                      0.34       0.83 f
  genoutpad/A (BUFC)                          0.00       0.83 f
  genoutpad/Z (BUFC)                          0.36       1.19 f
  gen_out (out)                               0.00       1.19 f
  data arrival time                                      1.19

  clock clkout (fall edge)                    0.00       0.00
  clock network delay (ideal)                 0.68       0.68
  output external delay                      -1.00      -0.32
  data required time                                    -0.32
  ------------------------------------------------------------
  data required time                                    -0.32
  data arrival time                                     -1.19
  ------------------------------------------------------------
  slack (MET)                                            1.52
```

And where does the "0.68" clock network delay comes from? Well, that's the falling edge delay to clkout:

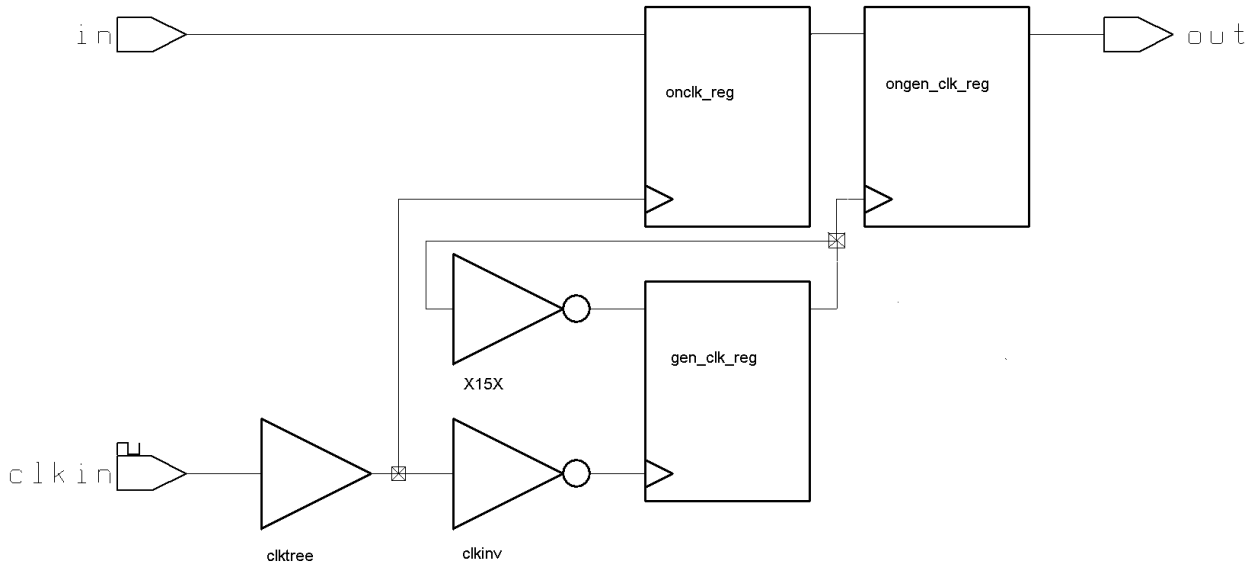
```
  report_timing -to [get_ports clkout] -delay max_fall

  Startpoint: gen_clk_reg/Q
              (clock source 'clkout_at_q')
  Endpoint: clkout (output port)
  Path Group: (none)
  Path Type: max

  Point                                       Incr       Path
  ------------------------------------------------------------
  clock source latency                        0.34       0.34
  gen_clk_reg/Q (FD1QA)                       0.00       0.34 r
  genclktree/Z (BUFC)                         0.15       0.49 r
  clkoutpad/Z (N1B)                           0.18       0.67 f
  clkout (out)                                0.00       0.68 f
  data arrival time                                      0.68
```

5.0 Describing complex clocking relationships using –edges

Most common divide_by clocks can be described using the "-divide_by" option on create_generated_clock as described above. There are cases, however, where this doesn't work, particularly if things are happening on falling edges.

5.1 Example 1: Basic use of –edges

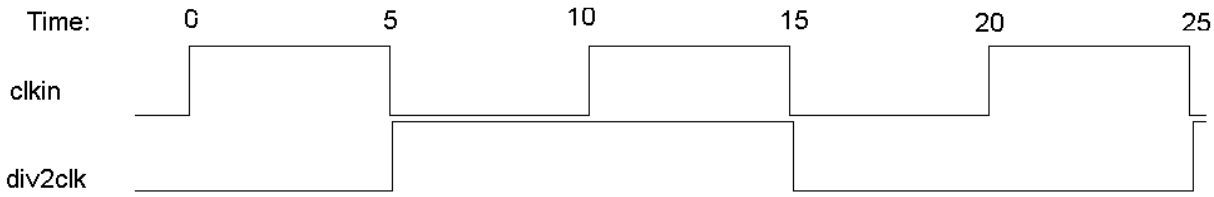
Here's a circuit which passes data from a flop clocked on clkin to a flop clocked on a divide-by-2 clock derived from clkin:



If we ignore the inverter clkinv in the path to the clock divider flop gen_clk_reg, this circuit is very like the one encountered earlier that had the divide-by-2 outgoing clock. You do a create_clock on clkin, and a create_generated_clock on gen_clk_reg/Q, and off you go.

But that inverter "clkinv" poses a problem (it is there, by the way, to avoid hold time problems associated with passing the data from the clkin domain to the gen_clk domain). You would hope that PrimeTime would see the inverter in the path to gen_clk_reg and handle this all automatically, and you'd be half right. PrimeTime will recognize the inversion when calculating the clock propagation value (it will time the falling edge through the buffer clktree, and the falling-to-rising edge through the inverter clkinv), but it will not recognize the inversion when it locates the edge for the timing check.

It might help to show the clock waveforms:



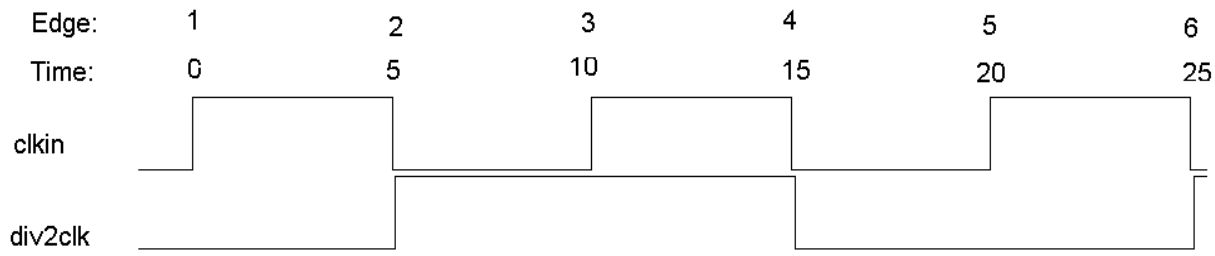
We'll get to the details of the script in a minute, but this is what you would get if you just did a normal create_generated_clock with a "-divide_by 2" option:

```
  report_timing -to [get_pins ongen_clk_reg/D] -input_pins -path_type
full_clock

  Startpoint: onclk_reg (rising edge-triggered flip-flop clocked by clkin)
  Endpoint: ongen_clk_reg
            (rising edge-triggered flip-flop clocked by div2clk)
  Path Group: div2clk
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clkin (rise edge)                 10.00      10.00
  clock source latency                     0.00      10.00
  clkin (in)                               0.00      10.00 r
  clktree/A (BUFC)                         0.00      10.00 r
  clktree/Z (BUFC)                         0.12      10.12 r
  onclk_reg/CP (FD1QA)                     0.00      10.12 r
  onclk_reg/CP (FD1QA)                     0.00      10.12 r
  onclk_reg/Q (FD1QA)                      0.34      10.45 f
  ongen_clk_reg/D (FD1QA)                  0.00      10.45 f
  data arrival time                                  10.45

  clock div2clk (rise edge)               20.00      20.00
  clock source latency                     0.55      20.55
  gen_clk_reg/Q (FD1QA)                    0.00      20.55 r
  ongen_clk_reg/CP (FD1QA)                 0.00      20.55 r
  library setup time                      -0.27      20.29
  data required time                                 20.29
  -----------------------------------------------------------
  data required time                                 20.29
  data arrival time                                 -10.45
  -----------------------------------------------------------
  slack (MET)                                         9.84
```

If you look closely, you'll see that this timing calculaton isn't correct. The rise edge of div2clk occurs on falling edges of clkin. So, that "20.0" value should be "15.0". Or the clkin edge should be 0 and the div2clk edge should be 5.0 Either way will give the correct calculation.

You might be tempted to use the "-invert" switch on create_generated_clock, but, as we saw earlier, this refers to inversion of the created clock itself, not the inversion of the source clock from which it is derived. So, how do you describe this to PrimeTime? You use the "-edges" switch.

To understand how to used "-edges", here's the clock waveform again with the edges marked as defined by PrimeTime:



As you can see, if the data was launched from rising clkin at time 0, it should be captured by div2clk at time 5. The "Edge:" value shown is how PrimeTime defines edges. So, div2clk rises on edge 2, falls on edge 4, and rises again on edge 6. So, the correct –edges argument would be "-edges {2 4 6}".

Here's what the complete script looks like:

```
create_clock -period 10.0 [get_ports clkin]
set_propagated_clock clkin

create_generated_clock \
    -name div2clk \
    -source [get_ports clkin] \
    -edges {2 4 6} \
    [get_pins gen_clk_reg/Q]
set_propagated_clock div2clk
```

And here are the (now correct) timing reports.

Here's the setup report:

```
  report_timing -to [get_pins ongen_clk_reg/D] -input_pins -path_type
full_clock

  Startpoint: onclk_reg (rising edge-triggered flip-flop clocked by clkin)
  Endpoint: ongen_clk_reg
            (rising edge-triggered flip-flop clocked by div2clk)
  Path Group: div2clk
  Path Type: max

  Point                                       Incr       Path
  ------------------------------------------------------------
  clock clkin (rise edge)                     0.00       0.00
  clock source latency                        0.00       0.00
  clkin (in)                                  0.00       0.00 r
  clktree/A (BUFC)                            0.00       0.00 r
  clktree/Z (BUFC)                            0.12       0.12 r
  onclk_reg/CP (FD1QA)                        0.00       0.12 r
  onclk_reg/CP (FD1QA)                        0.00       0.12 r
  onclk_reg/Q (FD1QA)                         0.34       0.45 f
  ongen_clk_reg/D (FD1QA)                     0.00       0.45 f
  data arrival time                                      0.45

  clock div2clk (rise edge)                   5.00       5.00
  clock source latency                        0.55       5.55
  gen_clk_reg/Q (FD1QA)                       0.00       5.55 r
  ongen_clk_reg/CP (FD1QA)                    0.00       5.55 r
  library setup time                         -0.27       5.29
  data required time                                     5.29
  ------------------------------------------------------------
  data required time                                     5.29
  data arrival time                                     -0.45
  ------------------------------------------------------------
  slack (MET)                                            4.84
```

And here's the hold report:

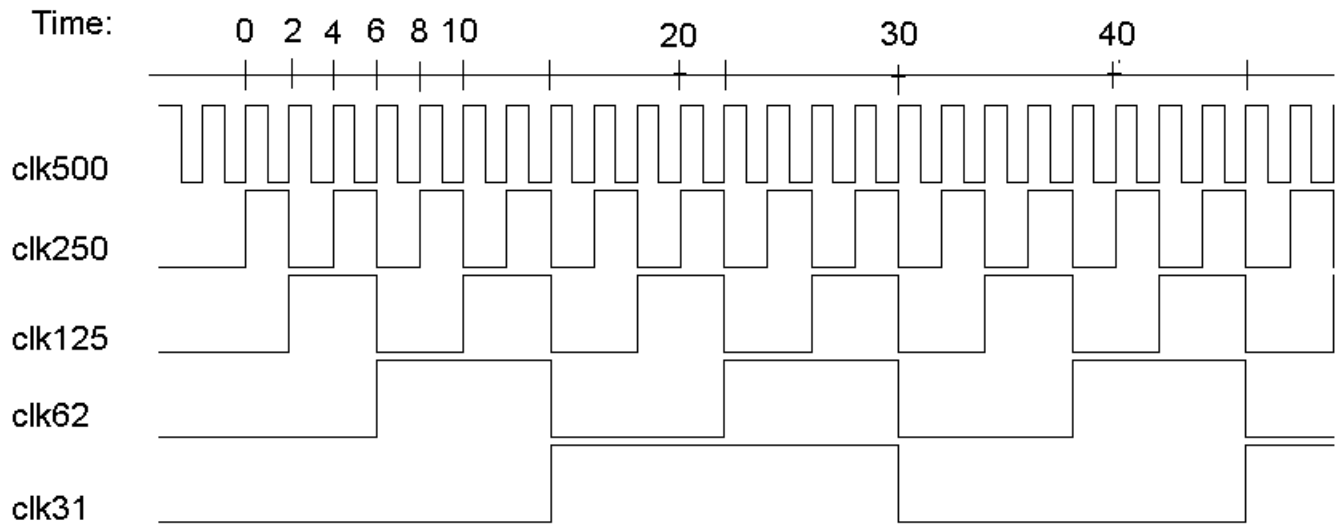
```
    report_timing -to [get_pins ongen_clk_reg/D] -input_pins -path_type
full_clock -delay min

  Startpoint: onclk_reg (rising edge-triggered flip-flop clocked by clkin)
  Endpoint: ongen_clk_reg
            (rising edge-triggered flip-flop clocked by div2clk)
  Path Group: div2clk
  Path Type: min

  Point                                       Incr       Path
  -------------------------------------------------------------
  clock clkin (rise edge)                    10.00      10.00
  clock source latency                        0.00      10.00
  clkin (in)                                  0.00      10.00 r
  clktree/A (BUFC)                            0.00      10.00 r
  clktree/Z (BUFC)                            0.12      10.12 r
  onclk_reg/CP (FD1QA)                        0.00      10.12 r
  onclk_reg/CP (FD1QA)                        0.00      10.12 r
  onclk_reg/Q (FD1QA)                         0.34      10.45 f
  ongen_clk_reg/D (FD1QA)                     0.00      10.45 f
  data arrival time                                     10.45

  clock div2clk (rise edge)                   5.00       5.00
  clock source latency                        0.55       5.55
  gen_clk_reg/Q (FD1QA)                       0.00       5.55 r
  ongen_clk_reg/CP (FD1QA)                    0.00       5.55 r
  ongen_clk_reg/CP (FD1QA)                               5.55 r
  library hold time                           0.16       5.72
  data required time                                     5.72
  -------------------------------------------------------------
  data required time                                     5.72
  data arrival time                                    -10.45
  -------------------------------------------------------------
  slack (MET)                                            4.74
```

5.2 Example 2: More complex use of –edges

Here's a more complex example. The following verilog code is a much simplified version of something I encountered in a recent design. It takes in a 500MHz clock, divides it down to 250MHz, then uses the falling edge of the 250MHz clock to clock a counter which generates the 125MHz, 62.5 MHz, and 31.25 MHz clocks from this.

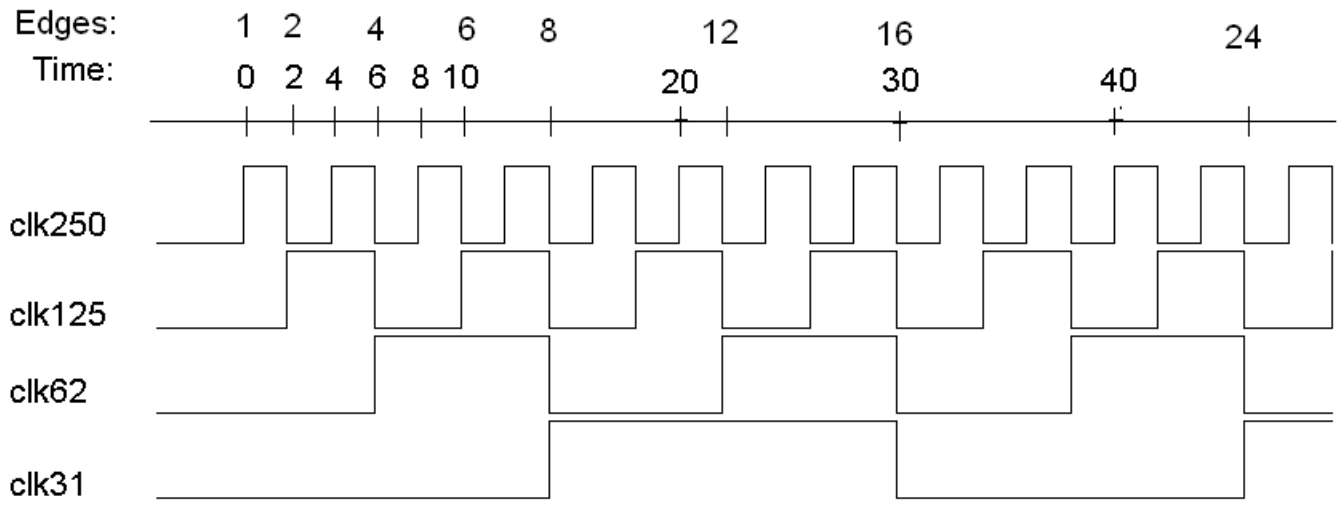
```
module edges(
clk500,
clk250,
clk125,
clk62,
clk31,
resetL
);

input       clk500;
output            clk250;
reg         clk250;
output            clk125;
reg         clk125;
output            clk62;
reg         clk62;
output            clk31;
reg         clk31;
input       resetL;


always @(posedge clk500 or negedge resetL)
begin
  if (!resetL)
    clk250 <= 0;
  else
    clk250 <= !clk250;
end

always @(negedge clk250 or negedge resetL)  // Note the use of negedge!
begin
  if (!resetL)
    {clk31,clk62,clk125} <= 0;
  else
    {clk31,clk62,clk125} <= {clk31,clk62,clk125} + 1;
end

endmodule
```

Here's what the timing waveform looks like:



Notice that each clock's rising transition occurs at the falling transition of its parent clock. No two rising edges happen at the same time. This splits the available clock period evenly between setup and hold when transferring data between clock domains.

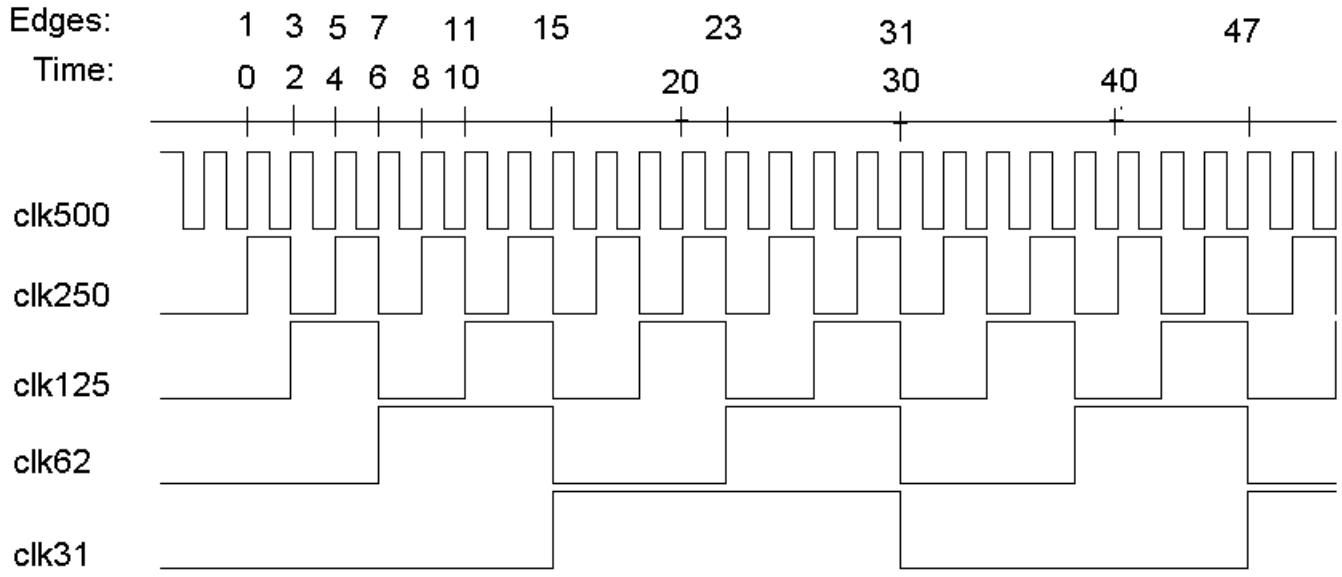
What's weird about this is that the clocks from 125MHz down are generated, not from the original 500 MHz source clock, but from the falling edge of the 250MHz clock. The first time I did this, I did create_generated_clock with "-divide_by 2" for clk250, then used the following map to find the "-edges" values for the other clocks:



Which results in the following mapping:

```
clk125  {2 4 6}
clk62   {6 8 12}
clk31   {8 16 24}
```

But this is wrong! The problem is that clk250 is itself a generated clock. As I pointed out earlier, all generated clocks need to be referenced to the original, non-generated source. In this case, that's clk500. So, here's what the new timing edge mapping looks like:



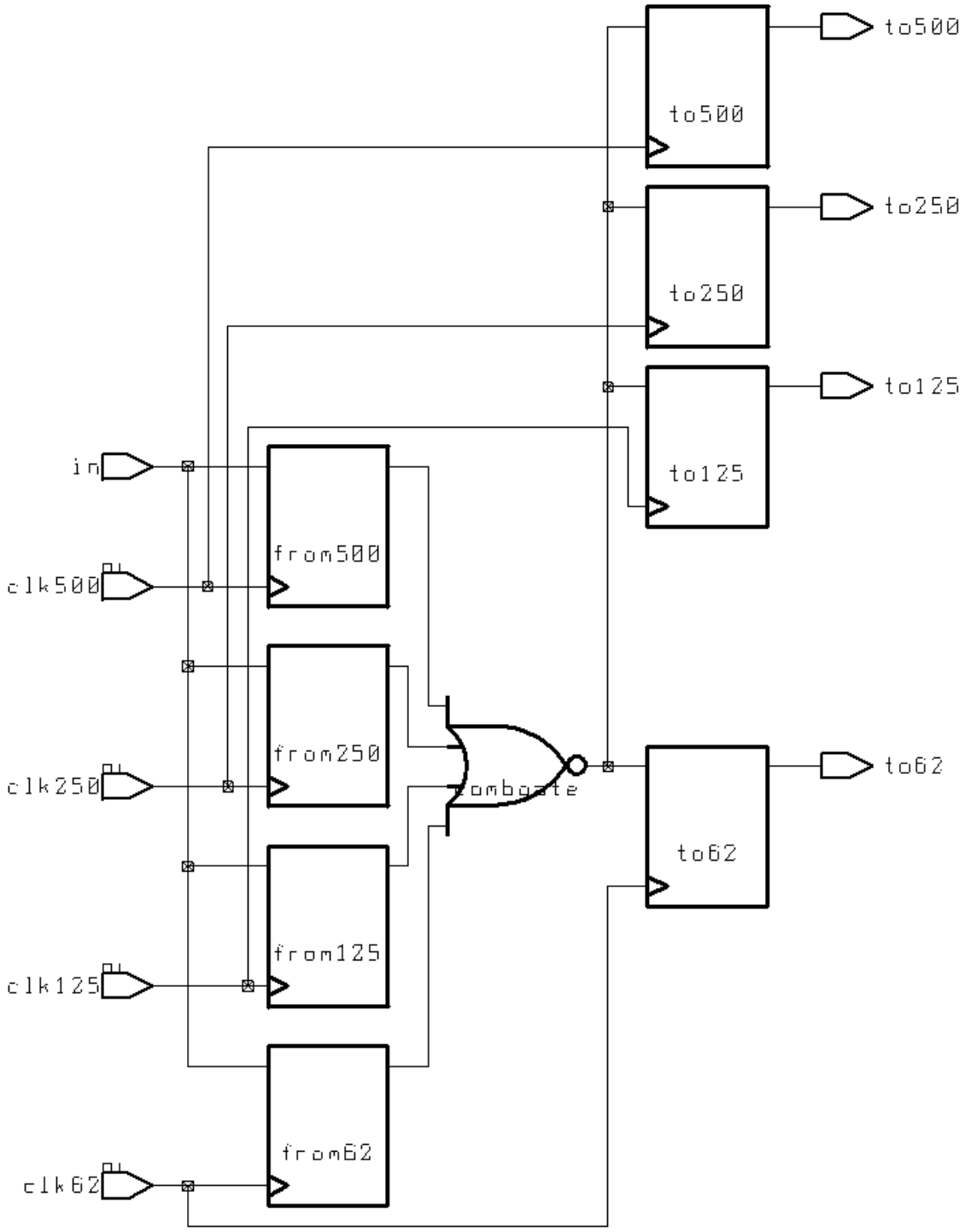
Which results in the following mapping:

```
clk125  {3 7 11}
clk62   {7 15 23}
clk31   {15 31 47}
```

And here's the PrimeTime code:

```
create_clock -period 2.0 [get_ports clk500]
set_propagated_clock clk500

# create the clk250 on the Q pin using -divide_by
create_generated_clock \
    -name clk250 \
    -source [get_ports clk500] \
    -divide_by 2 \
    [get_pins edges/clk250_reg/Q]
set_propagated_clock clk250

# create the divided clocks on the Q pins using -edges
create_generated_clock \
    -name clk125 \
    -source [get_ports clk500] \
    -edges {3 7 11} \
    [get_pins edges/clk125_reg/Q]
set_propagated_clock clk125
create_generated_clock \
    -name clk62 \
    -source [get_ports clk500] \
    -edges {7 15 23} \
    [get_pins edges/clk62_reg/Q]
set_propagated_clock clk62
create_generated_clock \
    -name clk31 \
    -source [get_ports clk500] \
    -edges {15 31 47} \
    [get_pins edges/clk31_reg/Q]
set_propagated_clock clk31
```

To prove that this works, I hooked it up to a little circuit that has a single input clocked by one flop from each clock domain. The outputs of these flops are then combined in a single combinational gate and the output of that gate is clocked by one flop from each clock domain again, like this:



I now have data paths from each clock domain to each other clock domain to generate reports.

Let's start with a basic report – from the clk250 back to clk250:

```
report_timing -from sampler/from250_reg/Q -to sampler/to250_reg/D

Startpoint: sampler/from250_reg
            (rising edge-triggered flip-flop clocked by clk250)
Endpoint: sampler/to250_reg
          (rising edge-triggered flip-flop clocked by clk250)
Path Group: clk250
Path Type: max

Point                                       Incr       Path
-----------------------------------------------------------
clock clk250 (rise edge)                    0.00       0.00
clock network delay (propagated)            0.44       0.44
sampler/from250_reg/CP (FD1QA)              0.00       0.44 r
sampler/from250_reg/Q (FD1QA) <-            0.37       0.81 f
sampler/combgate/Z (NR4A)                   0.44       1.25 r
sampler/to250_reg/D (FD1QA)                 0.00       1.25 r
data arrival time                                      1.25

clock clk250 (rise edge)                    4.00       4.00
clock network delay (propagated)            0.44       4.44
sampler/to250_reg/CP (FD1QA)                           4.44 r
library setup time                         -0.28       4.16
data required time                                     4.16
-----------------------------------------------------------
data required time                                     4.16
data arrival time                                     -1.25
-----------------------------------------------------------
slack (MET)                                            2.91
```

This is pretty straightforward. Data is launched at time 0 and captured at the first rising edge of clk250, at time 4.0.

Now let's look at the timing report from clk125 back to clk125:

```
report_timing -from sampler/from125_reg/Q -to sampler/to125_reg/D

Startpoint: sampler/from125_reg
            (rising edge-triggered flip-flop clocked by clk125)
Endpoint: sampler/to125_reg
          (rising edge-triggered flip-flop clocked by clk125)
Path Group: clk125
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
clock clk125 (rise edge)                       2.00       2.00
clock network delay (propagated)               1.15       3.15
sampler/from125_reg/CP (FD1QA)                 0.00       3.15 r
sampler/from125_reg/Q (FD1QA) <-               0.38       3.53 f
sampler/combgate/Z (NR4A)                      0.41       3.94 r
sampler/to125_reg/D (FD1QA)                    0.00       3.94 r
data arrival time                                         3.94

clock clk125 (rise edge)                      10.00      10.00
clock network delay (propagated)               1.15      11.15
sampler/to125_reg/CP (FD1QA)                             11.15 r
library setup time                            -0.27      10.88
data required time                                       10.88
-----------------------------------------------------------------
data required time                                       10.88
data arrival time                                        -3.94
-----------------------------------------------------------------
slack (MET)                                               6.94
```

Instead of launching at 0 and checking at 8.0, it launches at 2.0 and checks at 10.0. The net result of the calculation is, of course, the same. But our "-edges" switch as shifted clk125 over so that it has the correct phase relationship with the other clocks.

This is necessary in order to correctly calculate the cross-clock paths. Consider this path:

```
  report_timing -from sampler/from125_reg/Q -to sampler/to62_reg/D

Startpoint: sampler/from125_reg
            (rising edge-triggered flip-flop clocked by clk125)
Endpoint: sampler/to62_reg
          (rising edge-triggered flip-flop clocked by clk62)
Path Group: clk62
Path Type: max

Point                                       Incr       Path
-----------------------------------------------------------
clock clk125 (rise edge)                    2.00       2.00
clock network delay (propagated)            1.15       3.15
sampler/from125_reg/CP (FD1QA)              0.00       3.15 r
sampler/from125_reg/Q (FD1QA) <-            0.38       3.53 f
sampler/combgate/Z (NR4A)                   0.41       3.94 r
sampler/to62_reg/D (FD1QA)                  0.00       3.94 r
data arrival time                                      3.94

clock clk62 (rise edge)                     6.00       6.00
clock network delay (propagated)            1.10       7.10
sampler/to62_reg/CP (FD1QA)                            7.10 r
library setup time                         -0.27       6.83
data required time                                     6.83
-----------------------------------------------------------
data required time                                     6.83
data arrival time                                     -3.94
-----------------------------------------------------------
slack (MET)                                            2.88
```

This path is data launched by clk125 and captured by clk62. If you look at the timing diagram above, you'll see that the closest alignment of a clk125 rising edge to a clk62 rising edge if from time 2 .0 to time 6.0, which is what is being calculated above. Primetime will always look for the worst-case alignment of edges.

Now look at the "-delay min" (hold) report for this same path:

```
    report_timing -from sampler/from125_reg/Q -to sampler/to62_reg/D -delay
min

  Startpoint: sampler/from125_reg
              (rising edge-triggered flip-flop clocked by clk125)
  Endpoint: sampler/to62_reg
            (rising edge-triggered flip-flop clocked by clk62)
  Path Group: clk62
  Path Type: min

  Point                                     Incr       Path
  ------------------------------------------------------------
  clock clk125 (rise edge)                 10.00      10.00
  clock network delay (propagated)          1.15      11.15
  sampler/from125_reg/CP (FD1QA)            0.00      11.15 r
  sampler/from125_reg/Q (FD1QA) <-          0.38      11.53 r
  sampler/combgate/Z (NR4A)                 0.30      11.83 f
  sampler/to62_reg/D (FD1QA)                0.00      11.83 f
  data arrival time                                   11.83

  clock clk62 (rise edge)                   6.00       6.00
  clock network delay (propagated)          1.10       7.10
  sampler/to62_reg/CP (FD1QA)                          7.10 r
  library hold time                         0.17       7.27
  data required time                                   7.27
  ------------------------------------------------------------
  data required time                                   7.27
  data arrival time                                  -11.83
  ------------------------------------------------------------
  slack (MET)                                          4.56
```

Hold time reports with phase-shifted clocks are always a little hard to follow. Here's one way to look at it. We already saw that the setup path was from data launch at time 2.0 to data capture at time 6.0. This means that the data that changed at time 2.0 will be captured at time 6.0. When can it next change? Time 10.0. Therefore, the hold time calculation should start with the data changing at time 10.0, and check against capture at time 6.0 as in the timing trace above. You start out with a half-cycle of hold time margin – which is what this circuit was designed to do in the first place!

6.0 Miscellaneous tidbits

6.1 Phantom delays on input and output ports

Here's a common problem. You load up your netlist, read in the sdf, then report_timing from some input:

```
   report_timing -to f1_reg/D

Startpoint: in (input port)
Endpoint: f1_reg (rising edge-triggered flip-flop clocked by clkin)
Path Group: (none)
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
input external delay                     0.00       0.00 r
in (in)                                  0.00       0.00 r
inpad/Z (BUFB)                           1.14 *     1.14 r
f1_reg/D (FD1QA)                         0.10 *     1.24 r
data arrival time                                   1.24
```

You're *sure* that the input pad is faster than that, so you look in the sdf file, and, sure enough, it says the delay is 0.5ns. And yet the timing trace says "1.14". It has an "*" after it, so it *must* be backannotated, right?

Well, it turns out that if you read the PrimeTime documentation carefully, you'll find that the "*" means "all or part" is backannotated. This is one of my longstanding complaints. There's a *huge* difference between a delay being *all* backannotated and being *part* backannotated.

In this case, the "*" means "part". If you re-run the timing report with "-input_pins", you 'll see what is happening:

```
report_timing -to f1_reg/D  -input_pins

Startpoint: in (input port)
Endpoint: f1_reg (rising edge-triggered flip-flop clocked by clkin)
Path Group: (none)
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
input external delay                     0.00       0.00 r
in (in)                                  0.00       0.00 r
inpad/A (BUFB)                           0.64       0.64 r
inpad/Z (BUFB)                           0.50 *     1.14 r
f1_reg/D (FD1QA)                         0.10 *     1.24 r
data arrival time                                   1.24
```

The sdf's 0.5 ns delay through the input pad is there all right, but what is that "0.64" delay, and why doesn't it have an "*"?

That 0.64 delay is PrimeTime's *wireload-based estimate* of the delay on the "net" from input port in to pin inpad/A. There is no such net on the physical chip, of course (unless you count the bond wire), but PrimeTime doesn't know that. So, since no value was specified in the sdf file, PrimeTime estimates it for you. Since you're at the top of the chip (where the wireload tables are heavy), the value is likely to be large.

So, how do you get rid of it? You might try something like this:

```
set_annotated_delay 0.0 -net -from [get_ports in] -to [get_pins inpad/A]
```

And this would usually work. There's a subtle problem lurking, however. Suppose this design had a level of hierarchy containing the pad, such that the timing trace looked like this:

```
report_timing -to f1_reg/D -input_pins

Startpoint: in (input port)
Endpoint: f1_reg (rising edge-triggered flip-flop clocked by clkin)
Path Group: (none)
Path Type: max

Point                                    Incr       Path
-----------------------------------------------------------
input external delay                     0.00       0.00 r
in (in)                                  0.00       0.00 r
pads/in (pads)                           0.00       0.00 r
pads/inpad/A (BUFB)                      0.64       0.64 r
pads/inpad/Z (BUFB)                      0.50 *     1.14 r
pads/in_buf (pads)                       0.00 *     1.14 r
f1_reg/D (FD1QA)                         0.10 *     1.24 r
data arrival time                                   1.24
-----------------------------------------------------------
```

This is ok, you just do:

```
set_annotated_delay 0.0 -net -from [get_ports in] -to [get_pins pads/inpad/A]
```

The problem comes in when you try to automate this. The code would start out looking like this:

```
foreach_in_collection _input_pin [all_inputs] {
  set _input_net [all_connected $_input_pin]
  set _pad_input_pin \
    [remove_from_collection [all_connected $_input_net] $_input_pin ]
  set_annotated_delay -net 0.0 -from $_pad_output_pin -to $_output_pin
}
```

However, since PrimeTime doesn't do "all_connected" through the hierarchy, your set_annotated_delay is going to end up on the little chunk of net that goes from the port (in) to the hiearchical pin (pads/in). Oops.

I have discovered, however, that set_resistance *does* propagate through hierarchy, or, more accurately, ignores hierarchy. Here's what the output of the command:

```
report_net –connections –verbose [get_nets in]
```

looks like this before I do anything:

```
Connections for net 'in':
    pin capacitance:    0.0121
    wire capacitance:   1.12301
    total capacitance:  1.13511
    wire resistance:    0.56686
    number of drivers:  1
    number of loads:    1
    number of pins:     2

    Driver Pins             Type                Pin Cap
    ------------            ----------------    --------
    in                      Input Port          0

    Load Pins               Type                Pin Cap
    ------------            ----------------    --------
    pads/inpad/A            Input Pin (BUFB)    0.0121
```

It looks the same if I do:

```
report_net –connections –verbose [get_nets pads/in]

Connections for net 'pads/in':
    pin capacitance:    0.0121
    wire capacitance:   1.12301
    total capacitance:  1.13511
    wire resistance:    0.56686
    number of drivers:  1
    number of loads:    1
    number of pins:     2

    Driver Pins             Type                Pin Cap
    ------------            ----------------    --------
    in                      Input Port          0

    Load Pins               Type                Pin Cap
    ------------            ----------------    --------
    pads/inpad/A            Input Pin (BUFB)    0.0121
```

Now, if I do:

```
set_resistance 0.0 [get_nets in]
```

I get:

```
Connections for net 'in':
    pin capacitance:    0.0121
    wire capacitance:   1.12301
    total capacitance:  1.13511
    wire resistance:    0 (annotated)
    number of drivers:  1
    number of loads:    1
    number of pins:     2

    Driver Pins              Type                Pin Cap
    ------------             ----------------    --------
    in                       Input Port          0

    Load Pins                Type                Pin Cap
    ------------             ----------------    --------
    pads/inpad/A             Input Pin (BUFB)    0.0121
```

Note the “0 (annotated)” resistance. The key thing is that I get *exactly* the same trace if I do the report_net on pads/in:

```
Connections for net 'pads/in':
    pin capacitance:    0.0121
    wire capacitance:   1.12301
    total capacitance:  1.13511
    wire resistance:    0 (annotated)
    number of drivers:  1
    number of loads:    1
    number of pins:     2

    Driver Pins              Type                Pin Cap
    ------------             ----------------    --------
    in                       Input Port          0

    Load Pins                Type                Pin Cap
    ------------             ----------------    --------
    pads/inpad/A             Input Pin (BUFB)    0.0121
```

And this fixes the unannotated timing problem:

```
  report_timing -to f1_reg/D -input_pins

  Startpoint: in (input port)
  Endpoint: f1_reg (rising edge-triggered flip-flop clocked by clkin)
  Path Group: (none)
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 r
  in (in)                                  0.00       0.00 r
  pads/in (pads)                           0.00       0.00 r
  pads/inpad/A (BUFB)                      0.00       0.00 r
  pads/inpad/Z (BUFB)                      0.50 *     0.50 r
  pads/in_buf (pads)                       0.00 *     0.50 r
  f1_reg/D (FD1QA)                         0.10 *     0.60 r
  data arrival time                                   0.60
```

So, my script to fix the problem (for both inputs and outputs) looks like this:

```
foreach_in_collection _input_pin [all_inputs] {
  set _input_net [all_connected $_input_pin]
  set _pad_input_pin \
    [remove_from_collection [all_connected $_input_net] $_input_pin ]
  set_annotated_delay -net 0.0 -from $_input_pin -to $_pad_input_pin >
/dev/null
  set_resistance 0.0 $_input_net
}
foreach_in_collection _output_pin [all_outputs] {
  set _output_net [all_connected $_output_pin]
  set _pad_output_pin \
    [remove_from_collection [all_connected $_output_net] $_output_pin ]
  set_annotated_delay -net 0.0 -from $_pad_output_pin -to $_output_pin >
/dev/null
  set_resistance 0.0 $_output_net
}
```

I do the set_annotated_delay just to be sure. It probably isn't necessary.

6.2 Parsing command output

Going way back to the old design_time days, it seems like I always wanted some value that is only available in a report_*something* output, but not available directly for computation. With the advent of PrimeTime, many of these cases are now covered by get_timing_paths. But there are still cases where I want values other than those available with get_timing_paths.

Here's an example. Suppose that I have run check_timing, and there are flops not on any clock. I have reviewed these, and decided that they are ok in this mode. I don't want to have to inspect the report every time, so I want to get the number of flops not on any clock in a variable so I can check it against the expected value.

The basic approach goes back to a presentation given by Steve Golson at a long-ago SNUG entitled “My Favorite dc_shell Tricks”. What you do is to put the report output into a file, execute some shell script (via the “sh” command) to modify it to look like PrimeTime code (typically “set variable value”), then “source” the file.

In the bad old dc_shell language days, this was a right mess! It required an incredible number of “\” characters to escape things, and the resulting code was virtually unreadable. The introduction of tcl, and the use of perl instead of sed/awk/grep, makes things a little easier.

Here's my little script to get the check_timing results and stick them into an associative array called _check_timing_data

```
# default the array values
  set _check_timing_data("unconstrained_endpoints") 0
  set _check_timing_data("ignored_exceptions") 0
  set _check_timing_data("no_clock") 0
  set _check_timing_data("multiple_clocks") 0

  set _tempfile "temp[pid]"
  set _pt_tempfile "temp[pid].pt"

  set _cmd {#!/bin/sh
    cat <<END | perl -e '
    # Create an associative array mapping the LAST word of the warning
    # message to the _check_timing_data entry name.
    %tclnames = (
      delay => "unconstrained_endpoints",
      ignored => "ignored_exceptions",
      clock => "no_clock",
      clocks => "multiple_clocks",
    );

    # Loop over the output looking for lines that match:
    #   Warning: There <some_word> <count> <type of warning>
    # The <some_word> handles "is" or "are".
    while (<>) {
       if (/Warning: There \S+ (\S+) (.*)\.$/) {
         # Found one - same count and type of warning
         $value = $1;
         $type = $2;
         # Now get last word of <type of warning> and use it as an index
         # into the assoc array to fetch the corresponding name.
         @array = split(/ +/,$type);
         $lastword = $array[$#array];
         $tclname = "$tclnames{$lastword}";
         # And print out the "set" command
         print "set _check_timing_data(\"$tclname\") $value\n";
       }
     }
     '
  }

  #echo "cmd is $_cmd"

  # Build the executable file
  echo $_cmd > $_tempfile
  check_timing >> $_tempfile
  echo "END" >> $_tempfile
  # Make it executable and execute it.
  sh chmod +x $_tempfile
  sh ./$_tempfile > $_pt_tempfile

  # source the resulting output
  source $_pt_tempfile
```

Here's a quick explanation:

The first part defaults the array variables I intend to set, because check_timing only gives you a message if the value is non-zero, and tcl (unlike our dear friend perl) doesn't default to zero/null. So, I default all the array values to zero explicitly.

Next I use the "pid" to create unique temporary file names. The first temporary file will be the shell script to be executed. This will then create the ".pt" file that will be sourced.

Now comes the meat of the script. The "cat <<END" tells the shell interpreter to grab everything in the file up until it sees the word "END", and pipe it into the command "perl –e '….' ". I'm going to echo this out to the shell script file, then append the check_timing command output, then append the "END". That way, the output of check_timing is what gets piped into my little perl script. I put all of this into a variable ("set _cmd") because it avoids a lot of ugly escaping of special characters.

The perl script itself initializes an associative array called "names" which will map the last word of the check_timing message to my array entry name. The check_timing messages look like this:

```
Warning: There is 1 endpoint which is not constrained for maximum delay.
```

So, the perl script looks for lines that look like "Warning: There {some_word} …" and extracts the value (1 in this case) and the type "endpoint which is not constrained for maximum delay" in this case). The type is then split into an array and the last word extracted and used as a index into the names array to get the array entry name. The last command prints out something that looks like:

```
set _check_timing_data("unconstrained_endpoints") 1
```

into the ".pt" tempfile.

This perl code is echo'ed into the temp file, the check_timing output is appended, and then the "END" is appended. The temp file is made executable and executed. It creates the ".pt" file, which is then sourced, and off we go!

Finally, the temporary files are removed.

Here's what the first temp file (the shell script) looks like:

```
#!/bin/sh
    cat <<END | perl -e '
    # Create an associative array mapping the LAST word of the warning
    # message to the _check_timing_data entry name.
    %tclnames = (
      delay => "unconstrained_endpoints",
      ignored => "ignored_exceptions",
      clock => "no_clock",
      clocks => "multiple_clocks",
    );

    # Loop over the output looking for lines that match:
    #   Warning: There <some_word> <count> <type of warning>
    # The <some_word> handles "is" or "are".
    while (<>) {
       if (/Warning: There \S+ (\S+) (.*)\.$/) {
         # Found one - same count and type of warning
         $value = $1;
         $type = $2;
         # Now get last word of <type of warning> and use it as an index
         # into the assoc array to fetch the corresponding name.
         @array = split(/ +/,$type);
         $lastword = $array[$#array];
         $tclname = "$tclnames{$lastword}";
         # And print out the "set" command
         print "set _check_timing_data(\"$tclname\") $value\n";
       }
     }
     '

Warning: Some related clocks cannot be expanded to a common clock period
         within the expansion limit of 100 times per pair of related clock.
         The subject clocks are: clkin, div2clk, ... (PTE-033)
Warning: There is 1 endpoint which is not constrained for maximum delay.

0
END
```

The resulting .pt script you've already seen. It looks like this:

```
set _check_timing_data("unconstrained_endpoints") 1
```

This basic technique can be used to extract virtually any information from any of PrimeTime's commands using perl. And, once the basic technique is understood, this code is fairly easy to maintain – especially since the perl code just looks like perl code, without a lot of messy escape characters. As you can see from the example, you can even imbed comments.

6.0 Conclusion

Complex clocking situations involving muxes, divided clocks, falling edges, outgoing clocks, and combinations of these do occur in real chips, and PrimeTime can be used to time them correctly – if you know the tricks.