Consistent Timing Constraints with PrimeTime

Steve Golson

Trilobyte Systems
388 Stearns Street
Carlisle MA 01741
Phone: +1.978.369.9669
Fax: +1.978.371.9964
Email: sgolson@trilobyte.com
http://www.trilobyte.com

ABSTRACT

Physical implementation tools are usually *timing-driven*. They require *timing constraints* for reliable, repeatable, and successful operation. Generating and verifying these constraints is a familiar yet sometimes tedious task for the physical implementation engineer.

We introduce the idea of *consistent timing constraints* and show how PrimeTime can be used to create and manage timing constraint files needed for all other implementation tools, from synthesis to place-and-route to final chip finishing.

1.0 Introduction

Physical implementation means transforming an abstract RTL description of an integrated circuit into a physically-realizable representation of the same circuit. The primary rule for physical implementation engineers and their tools is *Do not change the functionality.* We go to great lengths to maintain and verify the functionality of our designs.

The second rule is *Meet the specified timing constraints*. These constraints are defined in the *data sheet* or *specification* for the design, in tabular and graphical form. But how are these constraints translated into a format usable by physical implementation tools?

This paper will delve into the problem of constraint generation for physical implementation tools. We introduce the concept of *consistent timing constraints* and discuss why this is critically important. We will outline a methodology relying on PrimeTime to generate and manage timing constraints, and to guarantee the consistency of your constraints, across all stages of the flow and all levels of hierarchy.

2.0 Problems: Management and Consistency

Many physical implementation tools are timing driven. A representative flow might include the following steps (not necessarily in this order), each of which requires timing constraints for proper operation:

automatic test pattern generation (ATPG)
cell placement
clock tree synthesis (CTS)
design-for-manufacturing/design-for-yield (DFM/DFY) tools
detail router
floorplanning
gate-level power analysis
global router
RTL power analysis
scan insertion
static timing analysis
synthesis from RTL to gate-level netlist
voltage drop analysis

These timing driven tools must have *correct* and *complete* constraints. *Correct* means the constraints used by the tool agree with the chip spec. *Complete* means you haven't left anything out—all timing paths have a constraint.

So you[1] might ask, what's the big deal? Just create a file, "the SDC file", edit it by hand, and feed it to all the tools.

1. Or your manager.

Unfortunately this doesn't work, due to differences amongst the tools:

File formats Although most tools can use Synopsys Design Constraints (SDC) format [1], some use Tcl scripts (*not* the same as SDC), and a few require custom formats.

Versions New commands are periodically added to the SDC spec. Your various tools might or might not support the latest commands.

Netlist status As the design proceeds through the flow, the timing constraints change. For example, prior to clock tree insertion the clocks are modeled as ideal nets, with specified network latency and uncertainty to model skew. After CTS the clocks can be propagated and the uncertainty is removed. Another example: scan clocks and I/Os must be constrained, but only after scan insertion. Another example: synthesis timing constraints cannot use library-specific pin names. Another example: synthesis constraints may be more conservative than signoff constraints (overconstrained synthesis).

Features Multi-corner multi-mode (MCMM) is the big new feature for implementation and analysis tools. Many tools do not have this capability yet.

Figure 1 shows a portion of a typical flow.

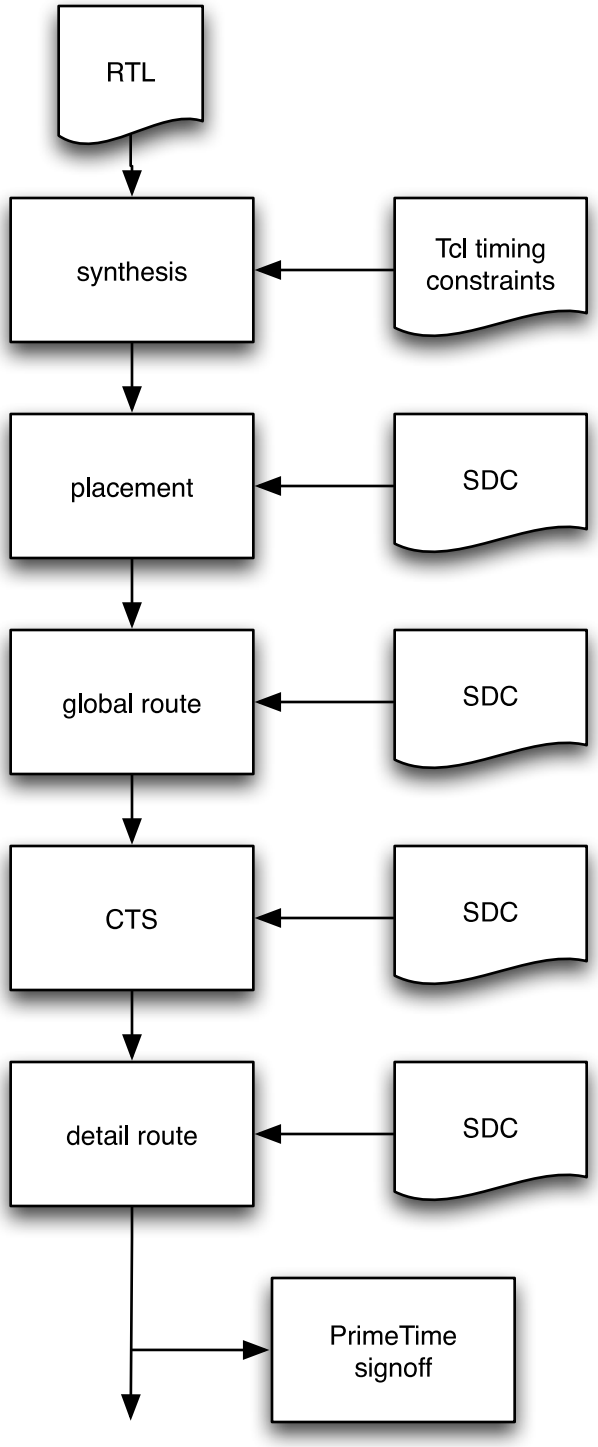

Figure 1. Typical flow

In summary, we have many tools used at various stages of the physical implementation flow. Each tool may require its own unique timing constraint file(s).

Problem #1: How shall we *manage* this multiplicity of files?

Problem #2: How can we make sure that each tool in turn is working on the same critical path? How do we ensure *consistency*?

3.0 Define: Consistency

Let's consider the second problem. As the design proceeds along the flow, each timing-driven tool in turn uses its particular constraints and internal timing engine. We've already shown that the constraint files will most likely not be *identical*. However the timing constraints must be *consistent*.

For example, after synthesis completes, run a timing report showing the critical paths in each clock group (path group). Now, moving along your flow, read the same netlist into the placement tool, and apply (possibly different!) timing constraints. Prior to doing anything with the placer, run a timing report. Compare with the synthesizer timing report. Do you see the same critical paths? Do you get the same slack?

This is important because we want every tool to be optimizing the design toward the same goal.

We define *consistency* as follows:

> Timing-driven tools have *consistent timing constraints* if, given the same netlist, they identify the same critical path for a given path group.

Here is a more practical definition. Consider timing-driven tools A and B. We say they have *consistent timing constraints* if for the same netlist

1. For tool A, for each path group, report the critical path.
2. Now in tool B, report the identical path (startpoint, endpoint, through points, clocks).
3. Tool B path must have same constraints (e.g., clock period, clock latency, multicycle path, input/output delay).
4. Tool B must report the same slack.

The converse must also be true (swap A and B).

Note you will have small timing differences due to different timing engines in the tools! This difference is hard to quantify; you must learn from experience on your particular design. Generally the differences are no more than a few percent of the total delay on the path. So let's modify our definition to read

4. Tool B must report the same slack, within some reasonable margin.

Note it's possible for some clock groups to be only in one tool. This depends on the multi-clock, multi-corner, multi-mode capabilities of the tools.

Note this only works for tools adjacent to each other in the flow. As the design progresses down the flow, the critical path and timings will most likely change.

4.0 Solution #2: Use PrimeTime throughout the flow

PrimeTime is normally used as a signoff tool, at the end of the flow. However your PrimeTime signoff Tcl scripts can be easily modified so they can be used at any point earlier in the flow. This allows us to use PrimeTime as the reference tool, and prove that each implementation tool in the flow is *consistent* with the signoff constraints.

Note we are not looking at the final timing reports from the signoff netlist—rather we are running PrimeTime on the intermediate netlists.

It is a straightforward task to modify signoff PrimeTime Tcl constraints such that they can be used at other points in the flow. There are a small number of switches required:

FIX_HFN (true or false): Early in the flow it is common for the synthesis tool to treat high fanout nets (HFNs) as ideal nets. The place&route tool is then used to build topologically-aware buffer trees. Also need integer variable **HFN_FANOUT_THRESHOLD** such that any net with fanout greater than this is given ideal timing.

PARASITICS_FILE (specified or not): If this file (or files) are specified then parasitics are annotated onto the design. Otherwise a wireload model is used. Note this could be estimated parasitics from Design Compiler Topographical or actual post-route extracted parasitics.

PROPAGATE_CLOCKS (true or false): Prior to clock tree synthesis (CTS) all clock nets are treated as ideal nets with specified network latency, and additional uncertainty is added to model clock skew. After CTS the clocks are propagated with actual skew.

ENABLE_SI (true or false): Crosstalk analysis is only needed for final signoff, otherwise disabled.

Netlist status (automatically determined): As mentioned before, there may be constraint changes due to netlist status, for example whether or not scan has been inserted. Rather than add additional switches, we can often determine the netlist status using various PrimeTime commands. For example it is straightforward to determine whether or not the scan clock port exists? and does it have a net attached to it? If not we assume scan is not inserted, thus scan clocks and associated constraints may be skipped.

Tcl code examples using these switches are given in the Appendix starting on page 13.

5.0 Solution #1: Use PrimeTime to generate all needed constraint files

Now the way is clear to solve the first problem. We've got PrimeTime running as the reference timing tool, to ensure consistency along the flow. Now we can use `write_sdc` to generate required files for the other tools. If necessary, the generated SDC is post-processed with simple scripts (e.g., Perl).

Actually no, it's not that simple, because we still have the tool differences outlined earlier. But it's not too hard to make this work, in an automated, flexible, consistent way.

Versions Assume your place&route tool needs SDC version 1.4. Then do

```
write_sdc filename.sdc -nosplit -version 1.4
```

PrimeTime will automatically convert newer constructs into compatible older commands. For example, clocks defined with `set_clock_groups` will be converted to appropriate `set_false_path` commands.

Features Depending on the capabilities of the target tool we might need to remove or relax some of the PrimeTime constraints. For example, consider a tool that does not support multi-clock constraints (i.e., more than one clock object on a given net or pin). In this case, remove the appropriate clocks prior to writing out the SDC[2]:

```
deactivate_clocks [get_clocks scan_clk]
remove_clock [get_clocks scan_clk]
write_sdc filename.sdc -nosplit
```

Note this does not affect the main PrimeTime analysis scripts.

Netlist status For some target tools we must postprocess the generated SDC. Consider the synthesis constraints used by Design Compiler. All other tools can assume a gate-level netlist, but synthesis starts with RTL. So an SDC construct like

```
set_multicycle_path 2 -from [get_pins blockA/myflop/CP]
```

will not work. The flop has not been mapped; the pin does not exist. What we need instead is

```
set_multicycle_path 2 -from [get_cells blockA/myflop]
```

A simple Perl script can convert the gate-level SDC constraint file into an RTL Tcl constraint file suitable for Design Compiler. See the example in Appendix section A.6 on page 20.

File formats Sometimes a post-processed SDC is not sufficient. In these cases you may need to write a custom PrimeTime reporting script that generates the proper file. For example, you may want to analyze the propagated clock latencies from a post-route netlist, calculate an equivalent

2. The `deactivate_clocks` and `activate_clocks` procs are available on SolvNet [3].

ideal network latency and uncertainty, then write these values into a Tcl script for use by PrimeTime or in future RTL compile runs with Design Compiler. See the example in Appendix section A.7 on page 22.

Figure 2 shows our typical flow, now using PrimeTime to generate consistent constraints for various tools.

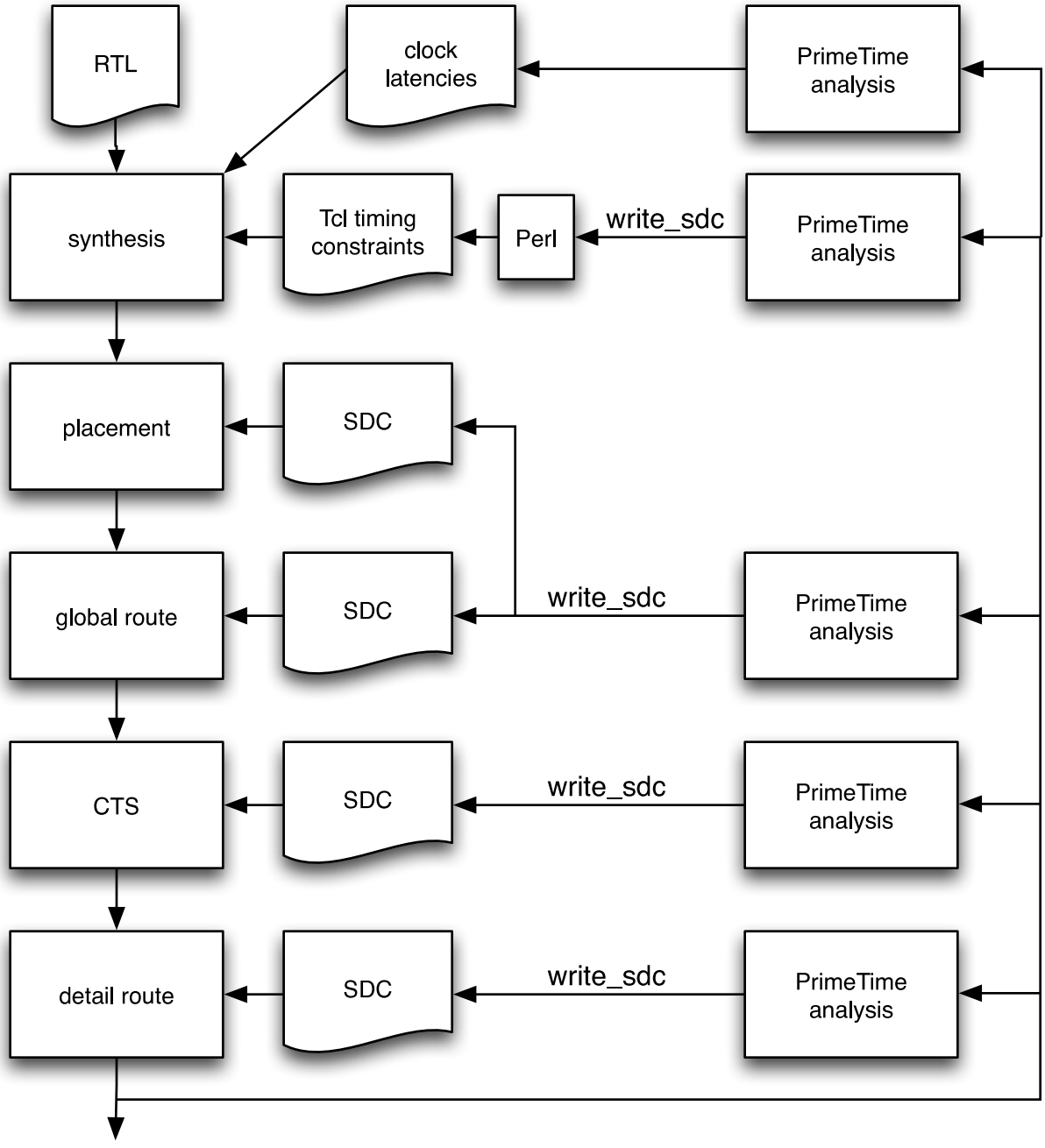

Figure 2. Typical flow using PrimeTime to generate consistent constraints

6.0 Problem: SDC command interpretation

The SDC specification [1] defines only syntax. It does not define the behavior of the commands. In particular it does not specify how to interpret ambiguous sequences of commands. As a result there may be differences in how the various EDA tools interpret the SDC commands. For example, consider the following three SDC commands:

```
set_multicycle_path 2 -from [get_clocks clkA]
set_multicycle_path 3 -from [get_clocks clkA] -to [get_clocks clkB]
set_multicycle_path 4                         -to [get_clocks clkB]
```

If I have a path from clkA to clkB, what is the correct multicycle path setting to use? Even if all tools are using the same version of SDC, these details of implementation and interpretation are not specified by the SDC spec. Your tools may implement this differently.

PrimeTime has a specific behavior that is discussed in the documentation for the various path exception commands. Here is an excerpt from the `set_multicycle_path` manpage:

> The `set_multicycle_path` command is a point-to-point timing exception command. The command can override the default single-cycle timing relationship for one or more timing paths. Other point-to-point timing exception commands include `set_max_delay`, `set_min_delay`, and `set_false_path`. False path information always takes precedence over multicycle path information. A specific `set_max_delay` or `set_min_delay` command overrides a general `set_multicycle_path` command.
>
> The more general commands apply to more than one path; either `-from` or `-to` (but not both), or clocks are used in the specification. Within a given point-to-point exception command, the more specific command overrides the more general. The following lists commands from highest to lowest precedence (more specific to more general):
>
> 1. `set_multicycle_path -from` *pin* `-to` *pin*
> 2. `set_multicycle_path -from` *pin* `-to` *clock*
> 3. `set_multicycle_path -from` *pin*
> 4. `set_multicycle_path -from` *clock* `-to` *pin*
> 5. `set_multicycle_path -to` *pin*
> 6. `set_multicycle_path -from` *clock* `-to` *clock*
> 7. `set_multicycle_path -from` *clock*
> 8. `set_multicycle_path -to` *clock*

Do your other EDA tools interpret these commands in the same way? If not, how will you manage this lack of consistency? Perhaps the discrepancies are in noncritical paths. Perhaps you can rewrite the exceptions so both tools interpret them the same way. It may be helpful to define special clock objects on just the clock pins of the affected flops, then use these clock objects to define the path exceptions.

7.0 Problem: Hierarchy and budgeting

Until now we have assumed all tools worked on the full chip. However for today's large designs it is common for many tools to operate on just a subset of your chip. These block-level tools require block-level timing constraints.

Nevertheless PrimeTime will be used for final full-chip signoff. In fact PrimeTime might be the only tool in the flow that looks at the entire chip. We still want to use our full-chip PrimeTime constraints to generate the files needed for the other block-level implementation tools.

So now we have another dimension to the problem of consistency. In addition to constraints that are consistent along the flow, we need constraints that are consistent across the hierarchy. We modify our definition of consistency as follows. Consider timing-driven tools A looking at a hierarchical block, and tool B looking at full-chip. We say they have consistent timing constraints if for the same netlist

1. For tool A, for each path group in the block, report the critical path.
2. Now in tool B, report the identical path (startpoint, endpoint, through points, clocks) at the full-chip level. If the path found in 1. starts and/or ends on a port of the block, then this port becomes a `-through` pin at the full-chip level.
3. Tool B full-chip path must have same constraints (e.g., clock period, clock latency, multicycle path, input/output delay).
4. Tool B must report the same slack, within some reasonable margin.

The converse must also be true (swap A and B). Again we note it's possible for some clock groups to be only in one tool. This depends on the multi-clock, multi-corner, multi-mode capabilities of the tools.

Creating consistent constraints across the hierarchy is a difficult problem. Furthermore, we must maintain these constraints in an automated fashion. As the design changes, and full-chip constraints are modified, the block-level constraints must be automatically updated. This is called *budgeting*.

Some floorplanning tools provide budgeting capabilities, but they are often only useful early in the design process, in an interactive and exploratory fashion. Thus these tools can help when putting together an initial *prototype flow* but cannot support a *production flow* later in the design process.

Even if your budgeting tool can be fully automated and used throughout the design process, many such tools only budget max delays (setup constraints). However the min delays (hold constraints) are critical as well, perhaps even more important. If only block-level implementation tools are used, then hold fixing must be done at the block level. This only works if we have reliable and consistent min delay constraints.

Another budgeting issue is that often some blocks are finished early and must be “locked down” prior to completion of the rest of the chip. The timing of these blocks is now fixed and should not be allocated during budgeting.

To avoid this difficult budgeting problem, often *default timing budgets* are used at the block level. Block-level path groups are created for each type of possible path: flop-to-flop, input-to-flop, flop-to-output, and input-to-output (or passthrough). Each path group can then be given default reasonable timing constraints. This greatly simplifies the constraint generation problem, and supports automation, but sacrifices *consistency.*

We can have default min delay budgets as well. This allows block-level hold fixing to be implemented in an automated way. However often these are very conservative hold constraints, and we end up doing much more hold fixing than is actually necessary. These unnecessary delay elements can waste significant extra area. Because the full-chip signoff timing passes the min delay checks, the area cost of these extra delay elements can be overlooked. To avoid this “silent killer” watch for excessive amount of positive min delay slack at the full-chip level, and keep track of the number of delay buffers inserted by your hold fixing tool.

Another typical problem with default timing budgets is the misprediction of clock latencies. This manifests itself after CTS when clock propagation is enabled. Block-level timing analysis will suddenly show failures on input-to-flop and flop-to-output paths, because the ideal clock latency used for `set_input_delay` and `set_output_delay` no longer agrees with the actual propagated clocks. This can be particularly bad with min delays and hold fixing. Sometimes users will go to great lengths to modify the block-level input and output delays timings after CTS, but the problem is actually due to incorrect clock latencies.

Perhaps a hybrid approach would work here. Default block-level budgets can specify input and output delays, but use full-chip postroute PrimeTime analysis to extract ideal clock latencies (min and max) and feed this information back to earlier block-level tools in the flow. After several iterations the design should stabilize and all tools at all levels of hierarchy will see the proper clock latencies in a consistent manner. See the Appendix for a PrimeTime script that extracts ideal clock latencies from postroute propagated clock networks.

Generating and maintaining block-level timing constraints is a huge data management problem. Do not underestimate the difficulty of this task.

8.0 Problem: Hierarchy and promotion

Sometimes we need to take block-level constraints and promote them upwards to be used at the full-chip level. Consider an IP block that will be part of our chip. Often such blocks are supplied with path exceptions: multicycle paths and false paths defined in an SDC file. Your full-chip PrimeTime scripts may be able to read these directly after executing an appropriate `current_instance` command. You might need to do some simple processing to these files prior to reading them in [2].

Consider a chip that contains multiple identical IP blocks (say, several copies of a CPU or IO interface). These blocks could easily have different block-level timing exceptions. They may be operating in different modes, or even at different voltages. Your methodology must support this—do not assume that “identical” blocks will have identical timing constraints.

9.0 Summary and Conclusions

Is this methodology a lot of work to implement? Yes, but once written and in place, the automation works beautifully. Also you can re-use much of this infrastructure on future projects; the required scripts are not design-specific.

Consider the alternative. It is possible to maintain all timing constraint files for all the tools by hand. In the early stages of a design project this is often the solution. It’s easy at first, but soon as more tools are added and as the timing constraints grow in complexity, this solution quickly becomes untenable.

Our goal instead is a solution that is

- automated: generate required files with no extra hand-editing.
- flexible: new tools can be added easily.
- consistent: constraints are consistent along the flow and across hierarchy.

We want repeatable, reliable results from our tools. This can only happen if you have *consistent* constraints. If one step in the flow has incorrect or incompatible constraints then the timing of your design will likely be irrepairably damaged, requiring much hand-fixing and tweaking. You cannot lie to your tools and expect useful results.

Also this methodology allows us to perform static timing analysis at various stages of the flow: post-synthesis, post-scan insertion, post-placement, post-route, post-CTS, etc.

In summary, we wish to manually edit and maintain the chip constraints in *one* place for *one* tool: full-chip PrimeTime Tcl scripts. All other required files can be automatically generated.

10.0 Acknowledgements

Many thanks to my reviewer Mark Sprague of AMD for his observations, discussions, and endless patience. Thanks to Jerry Frenkil of Sequence Design for pointing out yet more tools that require timing constraints.

11.0 References

[1] “Synopsys Design Constraints (SDC).” [Online]. Available: http://www.synopsys.com/Community/Interoperability/Pages/TapinSDC.aspx

[2] “Reusing module exceptions at the chip level,” Synopsys, SolvNet Doc Id 901542, June 11, 2003. [Online]. Available: http://solvnet.synopsys.com/retrieve/901542.html

[3] “Incrementally adding/removing active clocks in PrimeTime,” Synopsys, SolvNet Doc Id 010735, February 13, 2004. [Online]. Available: http://solvnet.synopsys.com/retrieve/010735.html

A.0 Appendix

Here are various interesting code fragments for use with techniques outlined in this paper.

A.1 Importing environment variables

This Tcl command will import all UNIX environment variables and convert them into Tcl variables with the same name. This is most useful when placed at the beginning of your local `.synopsys_pt.setup` file. Environment variables can now be easily used to transmit options and filenames into your PrimeTime scripts.

```
# get environment variables into Tcl variables
foreach {name value} [array get env] { eval set $name \{$value\} ; }
catch {unset name value}
```

A.2 FIX_HFN

When true, FIX_HFN will cause high fanout nets to be given ideal timing. Such nets are often treated as ideal during synthesis, with the intention that a more physically-aware tool later in the flow can build a better buffer tree. Running PrimeTime analysis on such a netlist requires these nets to be given ideal timing. First define a proc so we can use this in multiple scripts

```
proc make_hfn_timing_ideal {HFN_FANOUT_THRESHOLD} {

  set all_nets [get_net -hier -top_net_of_hierarchical_group]
  set total_high_fanout_nets 0

  echo -n “Searching [sizeof_collection $all_nets] nets “
  echo    “looking for fanouts > $HFN_FANOUT_THRESHOLD”

  foreach_in_collection the_net $all_nets {

    set fanout [sizeof_collection \
                   [filter \
                        [get_pins -quiet -leaf -of_objects $the_net] \
                        “pin_direction!=out”]]

    if {$fanout > $HFN_FANOUT_THRESHOLD} {

      if {[regexp {\*Logic[01]\*} [get_attribute $the_net base_name]]} {
        # this is a tie-off net, so no need to change it
        continue
      }

      # find all the pins on the net
      set all_pins  [get_pins -leaf -of_objects $the_net]
      # find all the pins driving the net
      set driver_pins [filter \
                           [get_pins -quiet -leaf -of_objects $the_net] \
                           “pin_direction==out”]
      # find all the ports driving the net
      set driver_ports [filter \
                            [get_ports -quiet -of_objects $the_net] \
                            “port_direction==in”]
      # report the drivers
      foreach_in_collection the_pin $driver_pins {
        echo -n “Setting Ideal Net on [get_object_name $the_net] with fanout $fanout”
        echo “ driven by pin [get_attribute $the_pin full_name]”
        set_ideal_network -no_propagate $driver_pins
      }
      foreach_in_collection the_port $driver_ports {
        echo -n “Setting Ideal Net on [get_object_name $the_net] with fanout $fanout”
        echo “ driven by port [get_attribute $the_port full_name]”
        set_ideal_network -no_propagate $the_port
      }
      incr total_high_fanout_nets
    }
  }
  echo “Fixed $total_high_fanout_nets nets with fanouts > $HFN_FANOUT_THRESHOLD”
}
```

and now you can invoke this in your analysis scripts

```
# fix high-fanout nets

if {$FIX_HFN} {
    make_hfn_timing_ideal $HFN_FANOUT_THRESHOLD
}
```

The HFN_FANOUT_THRESHOLD parameter defines the cutoff. This should agree with the value used in the synthesis scripts to control which nets are idealized during synthesis.

A.3 PARASITICS_FILE

Here is a simple PrimeTime Tcl proc to read a parasitics file. It determines the type of file by parsing the file name (SDF, SPEF, SBPF). Note this assumes a full-chip extraction; it is common to have more than one parasitics file corresponding to different hierarchical blocks. Adapting this procedure to such a complex scheme is left as an exercise.

```
proc read_parasitics_file {parasitics_file} {

  if {! [file exists ${parasitics_file}]} {
    return
  }

  if {[file extension $parasitics_file] == {.sdf}} {
    # this is an SDF file, so read it in and return
    echo Information: read_sdf ${parasitics_file}
    read_sdf ${parasitics_file}
    return
  }

  if $::ENABLE_SI {
    set si_args “-keep_capacitive_coupling”
  } else {
    set si_args “”
  }

  if {[file extension $parasitics_file] == {.sbpf}} {
    set type_args “-format SBPF”
  } else {
    # must be SPEF, so pick the right triplet value
    switch -regexp $::CORNER {
      {SLOW}    { set type_args “-triplet_type max” ; }
      {FAST}    { set type_args “-triplet_type min” ; }
      {TYPICAL} { set type_args “-triplet_type typ” ; }
      default   { set type_args “” ; }
    }
  }

  echo Information: read_parasitics ${si_args} ${type_args} -quiet ${parasitics_file}

  eval read_parasitics ${si_args} ${type_args} -quiet ${parasitics_file}

  # see what nets have not been annotated
  redirect ${::REPORT_DIR}/parasitics_not_annotated.rpt {
    report_annotated_parasitics -list_not_annotated -max_nets 10000
     }
}
```

The proc requires top-level variables REPORT_DIR, ENABLE_SI and CORNER.

A.4 PROPAGATE_CLOCKS

When true, PROPAGATE_CLOCKS causes all real clocks to be propagated. After clock tree synthesis has been performed, for correct timing you must propagate the actual clock latencies. The assumption is that ideal clock latencies have been set earlier in the analysis scripts, and we now selectively remove them.

```
# propagate the appropriate clocks

if {$PROPAGATE_CLOCKS} {
  # propagate all real clocks
  make_clocks_propagated [get_clocks -filter {defined(sources)}]
  set timing_remove_clock_reconvergence_pessimism true
} else {
  # no clocks are propagated
  # set_clock_transition 0 on real clocks
  make_clocks_ideal [get_clocks -filter {defined(sources)}]
}

list_ideal_clocks
list_propagated_clocks
```

Here are the procedures:

```
#-------------------------------------------------------------------------------
# make clocks propagated
#-------------------------------------------------------------------------------

proc make_clocks_propagated { {clocks “”} } {

    if {[string match “” $clocks]} {
        set clocks [get_clocks *]
    }

    # We must remove latency from all the possible places it could be defined.
    # Clock latency can be defined on a clock, pin, or port.
    # Thus we need to find all pins/ports in the transitive fanout of each clock.
    # We would like to do [get_attribute $the_clock clock_network_pins],
    # but that causes an implicit update_timing,
    # so instead use all_fanout which only needs a mini-update.

    # First find all the fanouts for all the clocks.
    # If you do the remove_clock_latency here as well then every time through
    # the loop you’ll do another update, because the previous clock changed
    # the timing by doing its own remove_clock_latency and set_propagated_clock.

    foreach_in_collection the_clock $clocks {
        set clock_name [get_object_name $the_clock]

        set fanout($clock_name) \
            [all_fanout -flat -from [get_attribute $the_clock sources]]
    }

    # Now make the clocks propagated

    foreach_in_collection the_clock $clocks {
        set clock_name [get_object_name $the_clock]

        remove_clock_latency -clock $clock_name $fanout($clock_name)
        remove_clock_latency                    $fanout($clock_name)
        remove_clock_latency         $the_clock

        # remove source latency on generated clocks only
        if {[string match {true} [get_attribute $the_clock is_generated]]} {
            remove_clock_latency -source $the_clock
        }

        set_propagated_clock $the_clock
    }
}

#-------------------------------------------------------------------------------
# make clocks ideal
#-------------------------------------------------------------------------------

proc make_clocks_ideal { {clocks “”} } {

    if {[string match “” $clocks]} {
        set clocks [get_clocks *]
    }
```

```
    # ideal clocks get 0 transition
    set_clock_transition 0 $clocks
}

#--------------------------------------------------------------------------------
#   report propagated clocks
#--------------------------------------------------------------------------------

proc get_propagated_clocks {args} {
    return [get_clocks -quiet -filter propagated_clock==true $args]
}

proc get_ideal_clocks {args} {
    return [get_clocks -quiet -filter propagated_clock==false $args]
}

proc list_propagated_clocks {} {
    set the_list {}
    foreach_in_collection dumb [get_propagated_clocks *] {
        lappend the_list [get_attribute $dumb full_name]
    }
    if {$the_list != {}} {
        echo “Information: propagated clocks are:”
        foreach i [lsort -dictionary $the_list] {
            echo “\t $i”
        }
    } else {
        echo “Information: No propagated clocks.”
    }
}

proc list_ideal_clocks {} {
    set the_list {}
    foreach_in_collection dumb [get_ideal_clocks *] {
        lappend the_list [get_attribute $dumb full_name]
    }
    if {$the_list != {}} {
        echo “Information: ideal (not propagated) clocks are:”
        foreach i [lsort -dictionary $the_list] {
            echo “\t $i”
        }
    } else {
        echo “Information: No ideal clocks. All clocks are propagated.”
    }
}
```

The useful procedures `deactivate_clocks` and `activate_clocks` are available for download from SolvNet [3].

A.5 ENABLE_SI

When true, ENABLE_SI causes signal integrity (crosstalk) analysis to be turned on. This variable is used in two places in the analysis script. First at the start of the script to ensure the license is available

```
if $ENABLE_SI {
    get_license PrimeTime-SI
    if {[get_message_info -occur SEC-50]} {
        echo Error: could not get PrimeTime-SI license
        exit 1
    }
}
```

then just before the main `update_timing` is performed we actually enable signal integrity analysis.

```
if $ENABLE_SI {
    echo Information: enabling PrimeTime-SI analysis
    set si_enable_analysis true
}
```

A.6 Converting SDC to Design Compiler Tcl constraints

This Perl script converts an SDC generated by PrimeTime `write_sdc` into a Design Compiler Tcl constraint file suitable for RTL synthesis. Path exceptions in the SDC are expressed using library-specific pin names which only exist after synthesis. For pre-synthesis constraints these are converted to cell names.

You will probably need to modify this to use the pin names for the flops in your synthesis library.

Similar scripts can be used to transform the PrimeTime SDC into an SDC suitable for tools that understand a more limited syntax. For example to create an SDC suitable for CeltIC you might delete `set_operating_conditions`, `set_wire_load_model`, and `set_max_transition` commands which cause a Warning in that tool.

```
#! /usr/bin/perl

# this script hacks up a PrimeTime SDC so it can be used for RTL synthesis
# the resulting script is Tcl, not SDC

#-------------------------------------------------------------------------
# get the original SDC filename

$input_sdc = $ARGV[0];

#-------------------------------------------------------------------------
# read in the sdc

open (FILE, "$input_sdc") or die "Could not open file $input_sdc : $!\n" ;

while (<FILE>) {

  # add comments to header
  if (/^# Created by PrimeTime write_sdc/) {
      print;
      print "# modified to Tcl for DC synthesis by fix_sdc_for_dc.pl\n";
      next;
    }

  # comment out SDC commands that don't work in DC Tcl
  if (/set sdc_version/) {
    s/^/# /;
  }
  if (/^set_operating_conditions/) {
    s/^/# /;
  }

  # skip case analysis that uses flop output pins
  next if (/set_case_analysis.*\[get_pins .*\/(Q|QN)(}| )/);

  # skip multicycle and false paths that use certain flop or latch pins
  next if (/set_(false|multicycle)_path.*\[get_pins .*\/(G|GN|RN|SN|SI|SE|E)(}| )/);

  # fix multicycle and false paths that use flop or latch pins
  # because the flops haven't been compiled yet, the pins don't exist
  # so change the pin names to cell names
  if (/set_(false|multicycle)_path.*\[get_pins .*\/(CK|CKN|D|Q|QN)(}| )/) {
    # need to convert the get_pins to get_cells,
    # but leave alone get_pins that are not flops
    s/get_pins( [^]]+\/(CK|CKN|D|Q|QN)(}| ))/get_cells\1/g;
    # need to convert the pin names to cell names
    s/(\S+)\/(CK|CKN|D|Q|QN)(}| )/\1\3/g;
  }

  print;
}

close(FILE);
```

A.7 Extracting clock latencies

In some cases the use of `write_sdc`, even with postprocessing, is not sufficient. For example consider the problem of clock latency. Early in the flow we must use ideal clock latencies, and clock skew is modeled using clock uncertainty. We would like an automated way to update these values once the actual propagated clock timings become known. Then as we continue iterating our flow, the tools early in the flow (e.g., Design Compiler) will use ideal clock latencies that accurately reflect the postroute physical results.

Here is a portion of the automatically generated clock latency Tcl script. Note that this does not contain Tcl commands, rather it defines variables. Each propagated clock in the design has a set of these variables defining the clock timing.

```
########## cpuA_clk
set clock_source_latency(cpuA_clk:SLOW:early:rise) 1.987
set clock_source_latency(cpuA_clk:SLOW:late:rise)  2.305
set clock_source_latency(cpuA_clk:SLOW:early:fall) 1.959
set clock_source_latency(cpuA_clk:SLOW:late:fall)  2.272
set clock_network_latency(cpuA_clk:SLOW:early:rise) 1.228
set clock_network_latency(cpuA_clk:SLOW:late:rise)  1.405
set clock_network_latency(cpuA_clk:SLOW:early:fall) 1.257
set clock_network_latency(cpuA_clk:SLOW:late:fall)  1.438
set clock_global_skew(cpuA_clk:SLOW)  0.187
```

Our Design Compiler synthesis script (or PrimeTime analysis script) can source these clock latency definition files:

```
source -echo -verbose ${SPEC_DIR}/mychip_clock_latencies.SLOW.tcl
source -echo -verbose ${SPEC_DIR}/mychip_clock_latencies.TYP.tcl
source -echo -verbose ${SPEC_DIR}/mychip_clock_latencies.FAST.tcl
```

There may be postroute clock objects which do not exist in the simpler synthesis constraints. Because these are variable definitions rather than Design Compiler Tcl (or SDC) commands, we only use the latency variables that correspond to clocks that exist. The others are defined but not used.

Not only Design Compiler, but PrimeTime also will use these latency definitions for full-chip timing analysis early in the flow.

Now set the ideal latencies and uncertainties for every existing clock. Note the analysis CORNER is defined as part of the variable index. Thus we have latency and uncertainty values for all corners available, which allows best-case/worst-case analysis if necessary.

```
# specify clock latency and uncertainty

foreach_in_collection the_clock [get_clocks] {

  set clock_name [get_object_name $the_clock]

  # source latency
  if [info exists clock_source_latency($clock_name:$CORNER:early:rise)] {
    set_clock_latency -min -early -rise -source \
        $clock_source_latency($clock_name:$CORNER:early:rise) $the_clock
    set_clock_latency -min -late  -rise -source \
        $clock_source_latency($clock_name:$CORNER:late:rise)  $the_clock
    set_clock_latency -min -early -fall -source \
        $clock_source_latency($clock_name:$CORNER:early:fall) $the_clock
    set_clock_latency -min -late  -fall -source \
        $clock_source_latency($clock_name:$CORNER:late:fall)  $the_clock
    set_clock_latency -max -early -rise -source \
        $clock_source_latency($clock_name:$CORNER:early:rise) $the_clock
    set_clock_latency -max -late  -rise -source \
        $clock_source_latency($clock_name:$CORNER:late:rise)  $the_clock
    set_clock_latency -max -early -fall -source \
        $clock_source_latency($clock_name:$CORNER:early:fall) $the_clock
    set_clock_latency -max -late  -fall -source \
        $clock_source_latency($clock_name:$CORNER:late:fall)  $the_clock
  }

  # network latency
  if [info exists clock_network_latency($clock_name:$CORNER:early:rise)] {
    set_clock_latency -min -rise \
        $clock_network_latency($clock_name:$CORNER:early:rise) $the_clock
    set_clock_latency -max -rise \
        $clock_network_latency($clock_name:$CORNER:late:rise)  $the_clock
    set_clock_latency -min -fall \
        $clock_network_latency($clock_name:$CORNER:early:fall) $the_clock
    set_clock_latency -max -fall \
        $clock_network_latency($clock_name:$CORNER:late:fall)  $the_clock
  }

  # simple clock uncertainty
  set_clock_uncertainty -setup [expr $setup_clock_extra_margin \
                                   + $setup_clock_OCV_margin \
                                   + $clock_local_skew($clock_name:$CORNER) \
                                  + $clock_additional_setup_uncertainty($clock_name)] \
                               $the_clock

  set_clock_uncertainty -hold  [expr $hold_clock_extra_margin \
                                   + $hold_clock_OCV_margin \
                                   + $clock_local_skew($clock_name:$CORNER) \
                                   + $clock_additional_hold_uncertainty($clock_name)] \
                               $the_clock
}
```

Here is the actual PrimeTime Tcl script which is run on the postroute full-chip design. All real propagated clocks will be analyzed and reported.

```
##########################################################################################

# issue warning if skew exceeds this value
set max_skew 0.600
# issue warning if early/late latencies differ by more than this percentage
set max_source_latency_diff_pct 15
set max_network_latency_diff_pct 15

##########################################################################
# define proc to warn about excessive variation

proc check_percent_difference {a b limit msg} {

  if {[expr $a > $b]} {
    set max $a
    set min $b
  } else {
    set min $a
    set max $b
  }

  if {$min == 0} {
    return
  }

  set percent [expr 100 * ($max - $min)/$min]

  if {$percent > $limit} {
    echo -n “Warning: $msg differ by [format {%.1f} $percent]% “
    echo    “which exceeds the limit of ${limit}%”
  }

  return
}

##########################################################################
# open the file for writing

set script_name ${REPORT_DIR}/clock_latencies.tcl
set channel [open $script_name w]

# output to file

puts $channel {##########################################################################}
puts $channel “# Written by write_clock_latencies.tcl”
puts $channel “# Running on [info hostname] with pid [pid] by [getenv USER]”
puts $channel “# [date]”
puts $channel “#”
puts $channel “# variable settings from analysis”
puts $channel “#”
puts $channel “# netlist_file          $netlist_file”
puts $channel “# parasitics_file       $parasitics_file”
puts $channel “# CORNER                $CORNER”
puts $channel “# MODE                  $MODE”
puts $channel “# FIX_HFN               $FIX_HFN”
```

```
puts $channel "# HFN_FANOUT_THRESHOLD  $HFN_FANOUT_THRESHOLD"
puts $channel "# PROPAGATE_CLOCKS      $PROPAGATE_CLOCKS"
puts $channel "# ENABLE_SI             $ENABLE_SI"
puts $channel {#####################################################################}

# loop over all the real propagated clocks

foreach_in_collection the_clock \
    [get_clocks -filter "defined(sources) && propagated_clock==true"] {

  set clock_name [get_object_name $the_clock]

  echo "##### Extracting latencies for clock $clock_name"
  echo ""

  puts $channel "########## $clock_name"

  ### get the source latencies (if any)

  set early_rise_source_latency \
      [get_attribute -quiet \
           [get_clocks $clock_name] clock_source_latency_early_rise_max]
  set late_rise_source_latency  \
      [get_attribute -quiet \
           [get_clocks $clock_name] clock_source_latency_late_rise_max]
  set early_fall_source_latency \
      [get_attribute -quiet \
           [get_clocks $clock_name] clock_source_latency_early_fall_max]
  set late_fall_source_latency  \
      [get_attribute -quiet \
           [get_clocks $clock_name] clock_source_latency_late_fall_max]

  if [string match {} $early_rise_source_latency] {
      set early_rise_source_latency 0 ;
  }
  if [string match {} $late_rise_source_latency]  {
      set late_rise_source_latency 0 ;
  }
  if [string match {} $early_fall_source_latency] {
      set early_fall_source_latency 0 ;
  }
  if [string match {} $late_fall_source_latency]  {
      set late_fall_source_latency 0 ;
  }

  # dump status to log file

  echo "source latency:"
  echo ""
  echo "  rise early $early_rise_source_latency"
  echo "       late  $late_rise_source_latency"
  check_percent_difference \
      $early_rise_source_latency \
      $late_rise_source_latency \
      $max_source_latency_diff_pct \
      "early rise and late rise source latency"
  echo ""
  echo "  fall early $early_fall_source_latency"
  echo "       late  $late_fall_source_latency"
```

```
  check_percent_difference \
      $early_fall_source_latency \
      $late_fall_source_latency \
      $max_source_latency_diff_pct \
      “early fall and late fall source latency”
  echo “”

  # print source latency values to Tcl script
  puts $channel “set clock_source_latency($clock_name:$CORNER:early:rise) [format
{%.3f} $early_rise_source_latency]”
  puts $channel “set clock_source_latency($clock_name:$CORNER:late:rise)  [format
{%.3f} $late_rise_source_latency]”
  puts $channel “set clock_source_latency($clock_name:$CORNER:early:fall) [format
{%.3f} $early_fall_source_latency]”
  puts $channel “set clock_source_latency($clock_name:$CORNER:late:fall)  [format
{%.3f} $late_fall_source_latency]”

  ### get the total latency

  set early_rise_time_list {}
  set early_fall_time_list {}
  set late_rise_time_list {}
  set late_fall_time_list {}

  foreach_in_collection clk_pin \
      [all_registers -edge_triggered -clock_pins -clock $clock_name] {

      # earliest clock latency is setup path capture clock
      set the_early_rise_time \
          [get_attribute -quiet \
               [get_timing_path \
                    -group $clock_name \
                    -delay max \
                    -to [get_pins \
                             -filter “direction==in && is_data_pin==true” \
                             -of [get_cells -of $clk_pin]]] \
               endpoint_clock_latency]
      set the_early_fall_time $the_early_rise_time
      # latest clock latency is hold path capture clock
      set the_late_rise_time  \
          [get_attribute -quiet \
               [get_timing_path \
                    -group $clock_name \
                    -delay min \
                    -to [get_pins \
                             -filter “direction==in && is_data_pin==true” \
                             -of [get_cells -of $clk_pin]]] \
               endpoint_clock_latency]
      set the_late_fall_time  $the_late_rise_time

      if [string match {} $the_early_rise_time] {
          # skip bogus ones
          echo Information: no early rise arrival time for clock $clock_name on pin
[get_object_name $clk_pin]
      } else {
          # subtract source latency to get network latency
          set the_early_rise_time [expr $the_early_rise_time -
```

```
$early_rise_source_latency]
          lappend early_rise_time_list $the_early_rise_time
      }

      if [string match {} $the_early_fall_time] {
          # skip bogus ones
          echo Information: no early fall arrival time for clock $clock_name on pin
[get_object_name $clk_pin]
      } else {
          # subtract source latency to get network latency
          set the_early_fall_time [expr $the_early_fall_time -
$early_fall_source_latency]
          lappend early_fall_time_list $the_early_fall_time
      }

      if [string match {} $the_late_rise_time] {
          # skip bogus ones
          echo Information: no late rise arrival time for clock $clock_name on pin
[get_object_name $clk_pin]
      } else {
          # subtract source latency to get network latency
          set the_late_rise_time [expr $the_late_rise_time -
$late_rise_source_latency]
          lappend late_rise_time_list $the_late_rise_time
      }

      if [string match {} $the_late_fall_time] {
          # skip bogus ones
          echo Information: no late fall arrival time for clock $clock_name on pin
[get_object_name $clk_pin]
      } else {
          # subtract source latency to get network latency
          set the_late_fall_time [expr $the_late_fall_time -
```

```
$late_fall_source_latency]
          lappend late_fall_time_list $the_late_fall_time
      }
  }

  # sort the lists

  set early_rise_time_list [lsort -real $early_rise_time_list]
  set early_fall_time_list [lsort -real $early_fall_time_list]
  set late_rise_time_list [lsort -real $late_rise_time_list]
  set late_fall_time_list [lsort -real $late_fall_time_list]

  set number_of_early_rise_values [llength $early_rise_time_list]
  set number_of_early_fall_values [llength $early_fall_time_list]
  set number_of_late_rise_values [llength $late_rise_time_list]
  set number_of_late_fall_values [llength $late_fall_time_list]

  if {($number_of_early_rise_values == 0) || \
      ($number_of_early_fall_values == 0) || \
      ($number_of_late_rise_values == 0) || \
      ($number_of_late_fall_values == 0)} {
      continue
  }

  # get the max, median, min values

  set late_rise_value(min)    [lindex $late_rise_time_list 0]
  set late_rise_value(5pct)   [lindex $late_rise_time_list [expr
int(ceil($number_of_late_rise_values * 0.05))]]
  set late_rise_value(median) [lindex $late_rise_time_list [expr
$number_of_late_rise_values / 2]]
  set late_rise_value(max)    [lindex $late_rise_time_list end]
  set late_rise_value(skew)   [expr $late_rise_value(max) - $late_rise_value(min)]
  set late_rise_value(5-100_skew) [expr $late_rise_value(max) -
$late_rise_value(5pct)]

  set late_fall_value(min)    [lindex $late_fall_time_list 0]
  set late_fall_value(5pct)   [lindex $late_fall_time_list [expr
int(ceil($number_of_late_fall_values * 0.05))]]
  set late_fall_value(median) [lindex $late_fall_time_list [expr
$number_of_late_fall_values / 2]]
  set late_fall_value(max)    [lindex $late_fall_time_list end]
  set late_fall_value(skew)   [expr $late_fall_value(max) - $late_fall_value(min)]
  set late_fall_value(5-100_skew) [expr $late_fall_value(max) -
$late_fall_value(5pct)]

  set early_rise_value(min)    [lindex $early_rise_time_list 0]
  set early_rise_value(5pct)   [lindex $early_rise_time_list [expr
int(ceil($number_of_early_rise_values * 0.05))]]
  set early_rise_value(median) [lindex $early_rise_time_list [expr
$number_of_early_rise_values / 2]]
  set early_rise_value(max)    [lindex $early_rise_time_list end]
  set early_rise_value(skew)   [expr $early_rise_value(max) - $early_rise_value(min)]
  set early_rise_value(5-100_skew) [expr $early_rise_value(max) -
$early_rise_value(5pct)]

  set early_fall_value(min)    [lindex $early_fall_time_list 0]
  set early_fall_value(5pct)   [lindex $early_fall_time_list [expr
```

```
int(ceil($number_of_early_fall_values * 0.05))]]
  set early_fall_value(median) [lindex $early_fall_time_list [expr
$number_of_early_fall_values / 2]]
  set early_fall_value(max)    [lindex $early_fall_time_list end]
  set early_fall_value(skew)   [expr $early_fall_value(max) - $early_fall_value(min)]
  set early_fall_value(5-100_skew) [expr $early_fall_value(max) -
$early_fall_value(5pct)]

  # dump status to log file

  echo “network latency:”
  echo ““
  echo “  rise early min     $early_rise_value(min)”
  echo “             5pct    $early_rise_value(5pct)”
  echo “             median  $early_rise_value(median)”
  echo “             max     $early_rise_value(max)”
  echo “             skew    $early_rise_value(skew)”
  echo “          5-100_skew $early_rise_value(5-100_skew)”
  echo “             #values $number_of_early_rise_values”
  if {$early_rise_value(5-100_skew) > $max_skew} { echo “Warning: exceeded max_skew of
$max_skew” ; }
  echo ““
  echo “       late  min     $late_rise_value(min)”
  echo “             5pct    $late_rise_value(5pct)”
  echo “             median  $late_rise_value(median)”
  echo “             max     $late_rise_value(max)”
  echo “             skew    $late_rise_value(skew)”
  echo “          5-100_skew $late_rise_value(5-100_skew)”
  echo “             #values $number_of_late_rise_values”
  if {$late_rise_value(5-100_skew) > $max_skew} { echo “Warning: exceeded max_skew of
$max_skew” ; }
  check_percent_difference $early_rise_value(median) $late_rise_value(median)
$max_network_latency_diff_pct “early rise and late rise network latency”
  echo ““
  echo “  fall early min     $early_fall_value(min)”
  echo “             5pct    $early_fall_value(5pct)”
  echo “             median  $early_fall_value(median)”
  echo “             max     $early_fall_value(max)”
  echo “             skew    $early_fall_value(skew)”
  echo “          5-100_skew $early_fall_value(5-100_skew)”
  echo “             #values $number_of_early_fall_values”
  if {$early_fall_value(5-100_skew) > $max_skew} { echo “Warning: exceeded max_skew of
$max_skew” ; }
  echo ““
  echo “       late  min     $late_fall_value(min)”
  echo “             5pct    $late_fall_value(5pct)”
  echo “             median  $late_fall_value(median)”
  echo “             max     $late_fall_value(max)”
  echo “             skew    $late_fall_value(skew)”
  echo “          5-100_skew $late_fall_value(5-100_skew)”
  echo “             #values $number_of_late_fall_values”
  if {$late_fall_value(5-100_skew) > $max_skew} { echo “Warning: exceeded max_skew of
$max_skew” ; }
  check_percent_difference $early_fall_value(median) $late_fall_value(median)
```

```
$max_network_latency_diff_pct “early fall and late fall network latency”
  echo “”

  # print network latency values to Tcl script

  puts $channel “set clock_network_latency($clock_name:$CORNER:early:rise) [format
{%.3f} $early_rise_value(median)]”
  puts $channel “set clock_network_latency($clock_name:$CORNER:late:rise)  [format
{%.3f} $late_rise_value(median)]”
  puts $channel “set clock_network_latency($clock_name:$CORNER:early:fall) [format
{%.3f} $early_fall_value(median)]”
  puts $channel “set clock_network_latency($clock_name:$CORNER:late:fall)  [format
{%.3f} $late_fall_value(median)]”

  # print skew value to Tcl script

  set global_skew [expr ( $early_rise_value(5-100_skew) + \
                          $late_rise_value(5-100_skew)  + \
                          $early_fall_value(5-100_skew) + \
                          $late_fall_value(5-100_skew) ) \
                      / 4.0 ]

  if {$global_skew > $max_skew} { set global_skew $max_skew ; }

  puts $channel “set clock_global_skew($clock_name:$CORNER)  [format {%.3f}
$global_skew]”
}

# close the file

puts $channel {###########################################################################}

close $channel

##########################################################################
# all done

print_message_info

##########################################################################
```

This script runs very quickly, even on large designs. Also note it does not need to be run on every iteration of the flow, but only when significant changes are made to the clock network. As the design converges, these extracted latency numbers will stabilize.

The script generates simple statistics for each clock. This provides early feedback that the clock network has a reasonable distribution with small skew.

```
##### Extracting latencies for clock cpuA_clk

source latency:

  rise early 1.987049
       late  2.304699
Warning: early rise and late rise source latency differ by 16.0% which exceeds the
limit of 15%

  fall early 1.958989
       late  2.272815
Warning: early fall and late fall source latency differ by 16.0% which exceeds the
limit of 15%

network latency:

  rise early min     1.123831
             5pct    1.143644
             median  1.228742
             max     1.316696
             skew    0.192388
          5-100_skew 0.173052
             #values 14493

       late  min     1.284190
             5pct    1.308831
             median  1.404956
             max     1.508652
             skew    0.224462
          5-100_skew 0.199858
             #values 14934

  fall early min     1.152224
             5pct    1.172550
             median  1.256856
             max     1.345621
             skew    0.192388
          5-100_skew 0.173052
             #values 14934

       late  min     1.316578
             5pct    1.341182
             median  1.437335
             max     1.541031
             skew    0.224462
          5-100_skew 0.199858
             #values 14934
```