TimeQuest User Guide

Wiki Release 1.1 December 9th 2010 By: Ryan Scoville

Introduction:

I have spent a good amount of time over the last few years helping designers with TimeQuest, and found myself writing emails and small documents explaining similar concepts over and over again. This includes answering questions on www.alteraforum.com under the user name rysc. This document is an effort to consolidate most of what I've learned about TimeQuest into a single source. It is a work in progress, and currently has significant sections missing. I hope to be updating this regularly, but am finding the more I enter, the more gaps there are. Right now the core information is there and has more than enough for most users. Looking at the page count, some might say there is too much information.

Recommendations:

1) Use the Bookmarks when viewing this document, to show the major points and allow for easy navigation. Examples seem to constantly require an explanation from another section. I added hyperlinks throughout the document, but I believe the Table of Contents/Bookmarks will help users navigate the content.

2) Read the first section, Getting Started. I tried to pack as much useful information that most designers need. Even if you have a good grasp on TimeQuest, it's probably worth a quick run through.

3) Read as much of this document as you can. Hopefully this helps the user get "the big picture" of static timing analysis, rather than only a small sub-section. User's that immediately jump to an example that is similar to their own often miss the many facets of static timing analysis, and are more likely to become frustrated or, worse yet, make mistakes.

4) Use TimeQuest. I've seen many users do the opposite of the last recommendation, where they pour over documentation, trying to understand every nuance of every sentence and screenshot before opening the tool. As much as I would like users to read everything, it's just as important to start adding SDC constraints to your design, running TimeQuest, and analyzing what happens. By the end of the Getting Started section, the user should have most of their core timing constraints entered, possibly some I/O constraints, and a good handle on timing analysis.

Contact:

TimeQuest support is not my primary responsibility, and so I will not be able to directly assist users. That being said, if there is anything ambiguous, incorrect, or missing, please contact me via www.alteraforum.com, sending an email to user Rysc. I also monitor the forum a good amount and will try to answer questions there, as I much prefer helping with issues on the forum rather than direct email, since it can hopefully help multiple users. If you post something and I don't respond, feel free to send me an email through the forum. That being said, if I am unable to respond, please don't be offended.

Table of Contents

SECTION 1: GETTING STARTED 5

QUARTUS SETUP 5
CORE TIMING 7
create_clock 7
derive_pll_clocks 9
derive_clock_uncertainty 10
set_clock_groups 10
Quick tip for writing set_clock_groups constraint 12
I/O TIMING 14
Step 1) Use create_clock to add a virtual clock for the I/O interface 14
Step 2) Add set_input_delay or set_output_delay on the I/O port/s 15
Step 3) Determine the default setup and hold relationship between the FPGA clock and virtual clock 16
Step 4) Add multicycles 18
Step 5) Modify the -max and -min delays to account for external delays. 19
ANALYZING RESULTS 22
The Iterative Methodology 23
A diving tool 25
report_timing 26
Correlating Constraints to the Timing Report 29

SECTION 2: TIMING ANALYSIS BASICS 31

BASICS OF SETUP, HOLD, RECOVERY AND REMOVAL 31
DEFAULT RELATIONSHIPS 35
Determining Default Setup and Hold Relationships in Three Steps. 35
Points of Interest for Default Relationships 41
Falling Edge Analysis 42
Periodicity 44
Relationships between Unrelated Clocks 45
Phase-Shift Affect on Setup and Hold 47
MULTICYCLES 48
Determining Multicycle Relationships in Five Steps 49
Multicycles - Two Common Cases 55
Case 1 - Opening the Window 55
Case 2 - Shifting the Window 57
MAX AND MIN DELAYS 58
The Dangers of set_max_delay and set_min_delay 59
Using set_max_delay and set_min_delay for Tsu, Th, Tco, Min Tco and Tpd 61
RECOVERY AND REMOVAL 66

SECTION 3: SDC CONSTRAINTS 73

CREATE_CLOCK 73
CREATE_GENERATED_CLOCK 74
How Generated Clocks are Analyzed 76
DERIVE_PLL_CLOCKS 77
DERIVE_CLOCK_UNCERTAINTY 78
DERIVE_CLOCKS 78
SET_CLOCK_GROUPS 78
SET_MULTICYCLE_PATH 80
GET_FANOUTS 83
SET_MAX_DELAY/SET_MIN_DELAY 84
SET_FALSE_PATH 85
SET_CLOCK_UNCERTAINTY 86

SET_CLOCK_LATENCY 86
SET_INPUT_DELAY/SET_OUTPUT_DELAY 87
SET_MAX_SKEW 89
CONSTRAINT PRIORITY 91
Priority between Different Constraints *92*
Priority between Equal Constraints *93*
Priority between Multiple Assignments to the Same Node *93*
Priority between Derived Assignments and User Assignments *94*

SECTION 4: THE TIMEQUEST GUI 96

ENTERING SDC CONSTRAINTS FROM THE GUI 96
GETTING STARTED - TIMING NETLISTS AND SDCS 99
MAJOR REPORTS 100
DEVICE SPECIFIC REPORTS 102
Report TCCS *102*
Report RSKM *102*
Report DDR *103*
Report Metastability *103*
REPORT_TIMING - IF YOU ONLY KNOW ONE COMMAND… 104
TQ_Analysis.tcl *104*
-false_path *106*
Path Filters *106*
DATASHEET REPORTS 107
Report Fmax *107*
Report Datasheet *107*
DIAGNOSTIC 109
report_clocks *109*
report_clock_transfers *109*
Report Unconstrained Paths - report_ucp *110*
report_sdc *111*
Report Ignored Constraints - "report_sdc -ignored" *113*
check_timing *113*
report_partitions *116*
CUSTOM REPORTS 116
Report Timing *116*
Report Minimum Pulse Width *116*
Report False Path *116*
Report Path/Report Net *116*
Report Exceptions *117*
Report Skew and Report Max Skew *117*
Report Bottlenecks *117*
Create Slack Histogram *118*
MACROS 119
Report All Summaries *119*
Report Top Failing Paths *119*
Report All I/O Timings *119*
Report All Core Timing *120*
Create All Clock Histograms *120*

SECTION 5: TIMING MODELS 121

WHY TIMING MODELS ARE IMPORTANT 121
TIMING MODELS 123
UNCERTAINTY 124
RISE/FALL VARIATION 124
UNATENESS 125
ON-DIE VARIATION 127

Common Clock Path Pessimism 128
SECTION 6: QUARTUS II AND TIMING CONSTRIANTS 132
SECTION 7: TCL SYNTAX FOR SDC AND ANALYSIS SCRIPTS 132
SECTION 8: COMMON STRUCTURES AND CIRCUITS 132
PLLs 132
Transceivers 132
LVDS 132
Memory Interfaces 132
Clock Muxes 132
Ripple Clocks 132
Clock Enables 132
SECTION 9: EXAMPLES 132
SECTION 10: MISCELLANEOUS 133
Strategies for False Paths 133
Analyzing Paths 133
Comparing set_input_delay/set_output_delay to Tsu/Th/Tco and min Tco 133

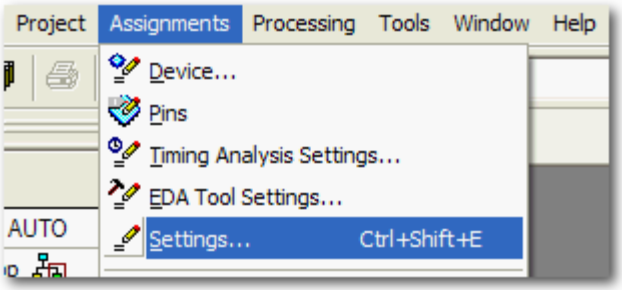
Section 1: Getting Started

This first section is meant to get a user up and running as quickly as possible. It touches on multiple topics that are detailed later, and is meant for application and a quick understanding. That being said, I think all users should look through this section and make sure they understand it.

The last portion of Getting Started covers analyzing results, which is an integral part of entering constraints. One can't enter core timing or I/O constraints without being able to read the analysis reports, so it is recommended to read that section in conjunction with the information at the beginning.

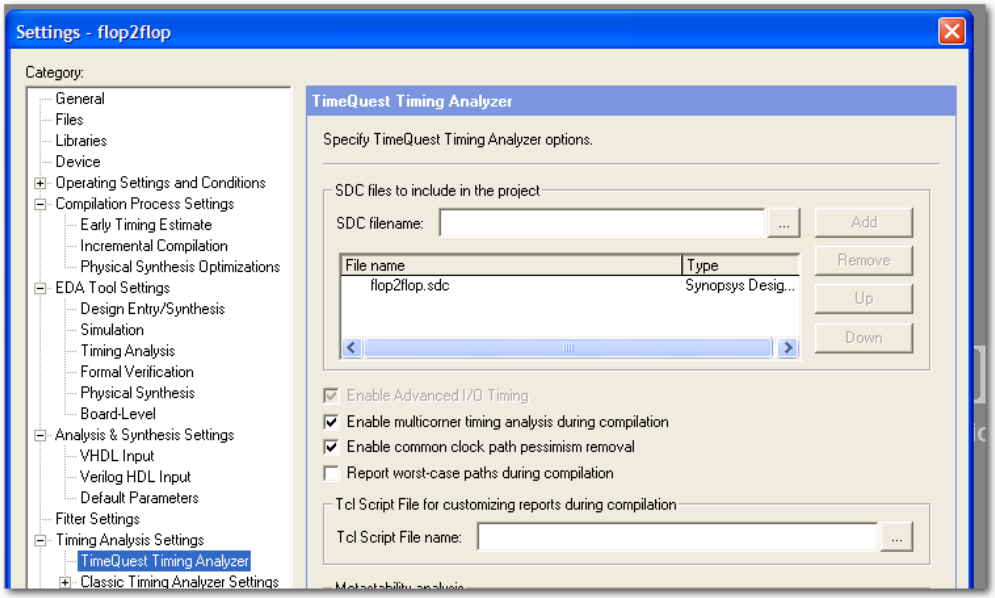
Quartus Setup

Within Quartus, there are a number of quick steps for setting up your design with TimeQuest. These are accessed through the pull-down menu Assignments -> Settings:



1) Along the left panel, select Timing Analysis Settings and select the "Use TimeQuest…" radio button. The Classic Timing Analyzer is the old timing analysis engine, which is not recommended for any new designs or architectures, and will eventually become obsolete.

2) Select TimeQuest Timing Analyzer in the left panel. The screenshot should look like below, whereby the user can add a new SDC file. SDC stands for Synopsys Design Constraint, which is the format TimeQuest uses, along with many other tools. If no .sdc file exists, we will create it in the next section. Note that SDC files are analyzed in the order listed, top to bottom.



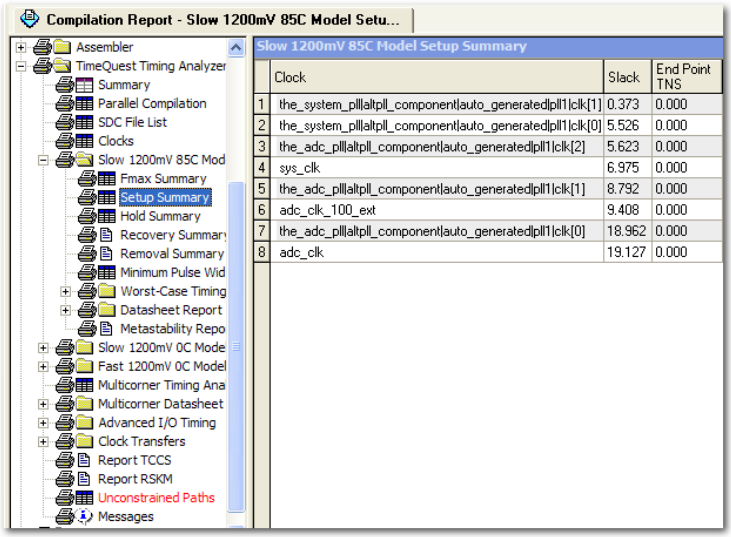
3) The following options are discussed more in the section on [Quartus II and Timing Constraints](#).

Check the following, which should be on by default:

- Enable multicorner timing analysis - This will analyze all the timing models of your FPGA against your constraints. This is required for final timing sign-off. Unchecked, only the slow timing model will be analyzed.
- Enable common clock path pessimism removal - Prevents timing analysis from over-calculating the effects of On-Die Variation. This makes timing better, and there really is no reason for this to be disabled.

Optional:

- Report worst-case paths during compilation. This option will show a summary of the worst paths in your Quartus report. We will be analyzing these paths in more detail in the TimeQuest tool. Some users like to see this summary up-front, but it also bloats the <project>.sta.rpt with all of these paths.
- Tcl script file for custom reports. We will use this later, adding custom reports for the user to run a custom analysis. For example, if the user is only working on a portion of the full FPGA, they may want additional timing reports that cover that hierarchy.



4) Simple, comprehensive static timing analysis summaries will be written to the Quartus II report during compilation. These reports cover the full analysis of **everything** constrained in the design. On a fully constrained design, these reports are enough to show if a design passes timing or not. The screenshot on the

left shows the setup slack to every clock domain in the design, and hence every setup analysis in the design is passing timing.

5) For more detailed analysis, the user must launch TimeQuest. Either go to the pull-down menu Tools -> TimeQuest Timing Analyzer, or click on the stopwatch icon in the Quartus II toolbar:

Core Timing

After compiling a project and launching TimeQuest, the user can now enter timing constraints. If no SDC file has been created, go to File -> New and create a new .sdc file. It can be saved with the same name as the project, and generally should be stored in the project directory.

Constraining the Core with Four Commands

Every beginning .sdc file should start with four components:

- *create_clock*
- *derive_pll_clocks*
- *derive_clock_uncertainty*
- *set_clock_groups*

The first three are almost trivial, and can get a user up and analyzing most of their design in a matter of minutes. As we go through these commands, be sure to look at The Iterative Method, which shows how to quickly modify .sdc files, re-run analysis, and keep iterating through more changes. Also, details about these commands can be found directly in TimeQuest by typing -long_help, such as:

create_clock -long_help
derive_pll_clocks -long_help
derive_clock_uncertainty -long_help
set_clock_groups -long_help

create_clock

When starting a new SDC file, the first thing to do is constrain the clocks coming into the FPGA with *create_clock*. The basic syntax looks like so:

create_clock -name sys_clk -period 8.0 [get_ports fpga_clk]

Notes:

- The above command creates a clock called sys_clk with an 8ns period and applies it to the port in the user's design called fpga_clk.

- Tcl and SDC are case-sensitive, so make sure fpga_clk matches the case used in your design.

- The clock will have a rising edge at time 0ns, and defaults to a 50% duty cycle, hence a falling edge at time 4ns. If the user wants a different duty cycle or to represent an offset, please use the -waveform option. This is very seldom necessary.

- Users often create a clock with the same name as the port it is applied to. This is perfectly legal. In the example above, this would be accomplished by:

create_clock -name fpga_clk -period 8.0 [get_ports fpga_clk]

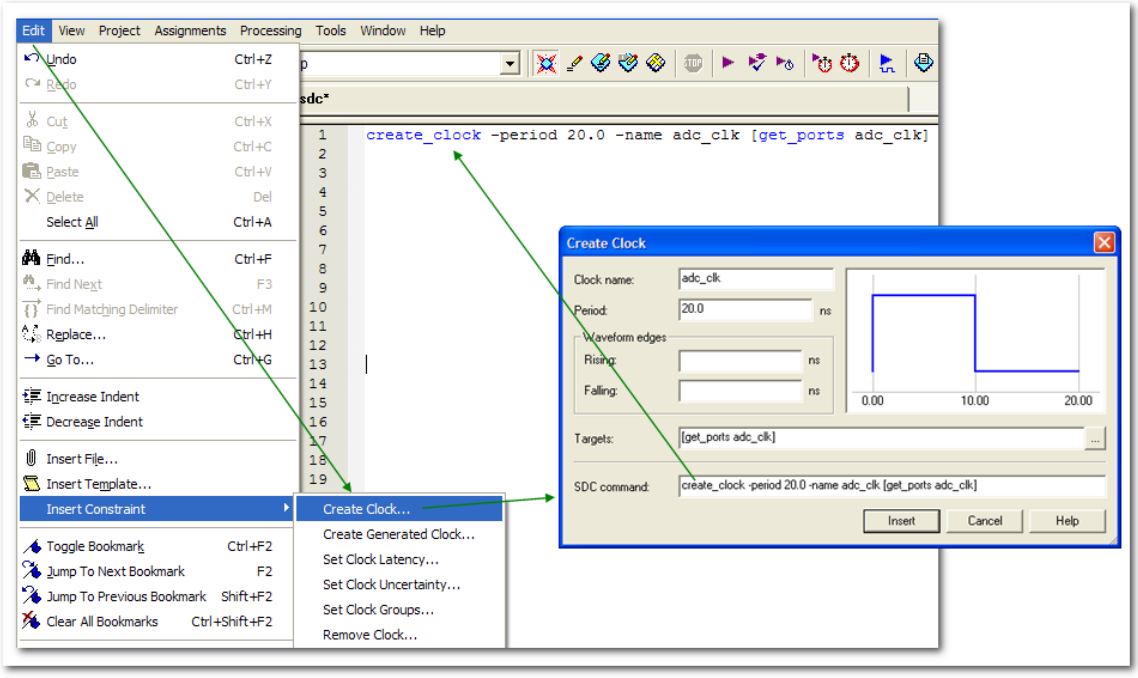
There are now two unique things called *fpga_clk*, a port in the user's design and a clock that emanates from that port.

- In Tcl syntax, square brackets will execute the command inside them, so [get_ports fpga_clk] will execute a command that finds all ports in the design that match fpga_clk and return them. This is discussed more in the Tcl syntax section. Although commonly used, many designers simply enter the port name like so:

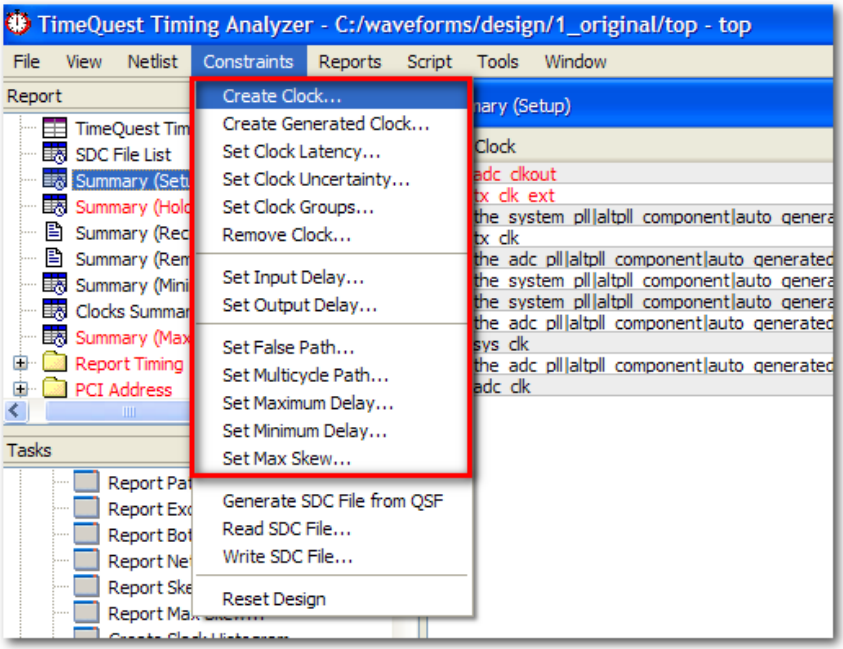
create_clock -name sys_clk -period 8.0 fpga_clk

- Repeat this step for all known clocks coming into the design. (If the user is unsure, just enter all the known clocks. Later on we will show how Report Unconstrained Paths can identify any unconstrained clocks).

Hint: Rather than typing constraints, users can enter constraints through the GUI. After launching TimeQuest, open the .sdc file from TimeQuest or Quartus II, place the cursor where the new constraint will go, and go to Edit -> Insert Constraint, and choose the constraint.



Also, DO NOT enter constraints from the TimeQuest GUI's Constraints pull-down menu:



Although it looks similar, these constraints will be applied directly to the timing database and not put into the .sdc file. Advanced users may find reasons to do this, but beginners should stay away from these and instead open the .sdc file and access them from Edit -> Insert Constraint.

derive_pll_clocks

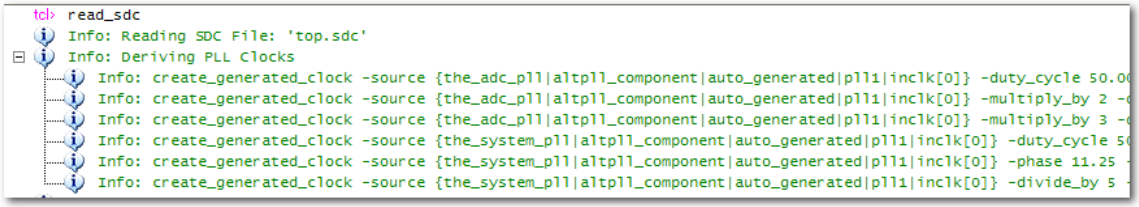
Add the following command into your .sdc:

derive_pll_clocks

That's it. Just type in that command.

Notes:

- Each output of a PLL should be constrained with *create_generated_clock*.
- When PLLs are created, the user enters how each PLL output is configured. Because of this, TimeQuest can auto-constraint them, which is what *derive_pll_clocks* is doing.
- This command does other useful things too. It constrains transceiver clocks. It adds multicycles between LVDS SERDES and user logic.
- To see all the low-level commands executed by *derive_pll_clocks*, the TimeQuest messages will explicitly show them as Info messages under Derive PLL Clocks:



- New designers often have the urge to not add *derive_pll_clocks*, and instead cut-and-paste each *create_generated_clock* assignment directly to the .sdc file. Technically there is nothing wrong with this, since the two are identical. The problem is that anytime a user modifies a PLL, they must remember to change the .sdc. Examples include modifying an existing output clock, adding a new PLL output, or making a change to the PLL's hierarchy. I have seen too many designers forget to modify their .sdc and spend time debugging something that *derive_pll_clocks* would have fixed automatically. My recommendation is to stick with *derive_pll_clocks.*

derive_clock_uncertainty

Add the following command to your .sdc:

derive_clock_uncertainty

Just type it in.

Notes:
- This should be in all SDC files for designs at 65nm and newer.
- It does not hurt to be in the .sdc file of older architectures, it just won't do anything.
- This command alculates clock to clock uncertainties within the FPGA, due to characteristics like PLL jitter, clock tree jitter, etc.
- A warning occurs if the user does not have this command in their .sdc.

Those are the first three steps, which can usually be done very quickly. For a sample design with two clocks coming into it, their .sdc might look like so:

```
create_clock -period 20.000 -name adc_clk [get_ports adc_clk]
create_clock -period 8.000 -name sys_clk [get_ports sys_clk]

derive_pll_clocks

derive_clock_uncertainty
```

set_clock_groups

With the constraints above, most if not all of the clocks in the design are now constrained. In TimeQuest, all clocks are related by default and it is up to the user to un-relate

clocks. So, for example, if there are paths between an 8ns clock and 10ns clock, even if the clocks are completely asynchronous, TimeQuest will see a 2ns setup relationship between these clocks and try to meet it. This is the conservative approach, in that TimeQuest analyzes everything known, rather than other tools which assume all clocks are unrelated and require the user to relate them. It is up to the user to tell TimeQuest which clocks are not related. The SDC language has a powerful constraint for doing this called set_clock_groups. The syntax, which may look complex at first is:

set_clock_groups -asynchronous -group {} -group {} -group {}

Notes:

- Each -group is a list of clocks that are related to each other
- There is no limit to the number of group options, i.e. *-group {}*. If a design needs fifty groups, that's fine. If entering the constraint through Edit -> Insert Constraint, it only has space for two groups, but this is only a limitation of that GUI. Feel free to manually add more into the .sdc file.
- User's look at the command and often think it is grouping clocks, but again, TimeQuest assumes all clocks are related, and so they're already in one big group. This command is really cutting timing between clocks in different groups within a set_clock_groups command.
- Any clock not listed in the assignment keeps the default of being related to all clocks, so if you forget a clock, it will conservatively be analyzed to all other domains it connects to.
- A clock cannot be within multiple -groups in a single assignment
- A user can have multiple set_clock_groups assignments
- This command is usually unreadable on a single line. Instead, make use of the Tcl escape character "\". By putting a space after your last character and then "\", the end-of-line character is escaped. (And be careful not to have any whitespace after the escape character, or else it will escape the whitespace, not the end-of-line character). The syntax for this will be shown shortly.
- For designs with complex clocking, writing this constraint can be an iterative process. For example, a design with two DDR3 cores and high-speed transceivers could easily have thirty or more clocks. In those cases, I just add the clocks I've created. Since clocks not in the command are still related to every clock, I am conservatively grouping what I know. If there are still failing paths in the design between unrelated clock domains, I start adding in the new clock domains as necessary. In this case, a large number of the clocks won't actually be in the set_clock_groups command, since they are either cut in the IP's .sdc file(like the ones generated by the DDR3 cores), or they only connect to clock domains they are related to.
- I generally leave virtual clocks created for I/O analysis out of this constraint. The only clocks they connect to are generally real paths, so there is no need to cut their analysis to other clocks.
- The option after *set_clock_groups* is either *-asynchronous* or *-exclusive*. The -asynchronous flag means the clocks are both toggling, but not in a way that can synchronously pass data. The -exclusive flag means the clocks do not toggle at the same time, and hence mutually exclusive. A good example of this might be a clock mux that has two generated clock assignments on its output. Since only one can toggle at a time, these clocks are *-exclusive*. TimeQuest will analyze your design identically for either flag. This option is really used for ASICs, that will analyze SI issues like cross-talk between clocks that are *-asynchronous*, but not

analyze cross-talk between clocks that are *-exclusive*. If going to Hardcopy, which uses ASIC analysis tools on the back-end, it is recommended to get this right. For FPGAs it really does not matter. The more conservative value is *-asynchronous*, since this states the clocks can interfere with each other, and what I use by default.

- Another way to cut timing between clocks is to use *set_false_path*. To cut timing between sys_clk and dsp_clk, a user might enter:

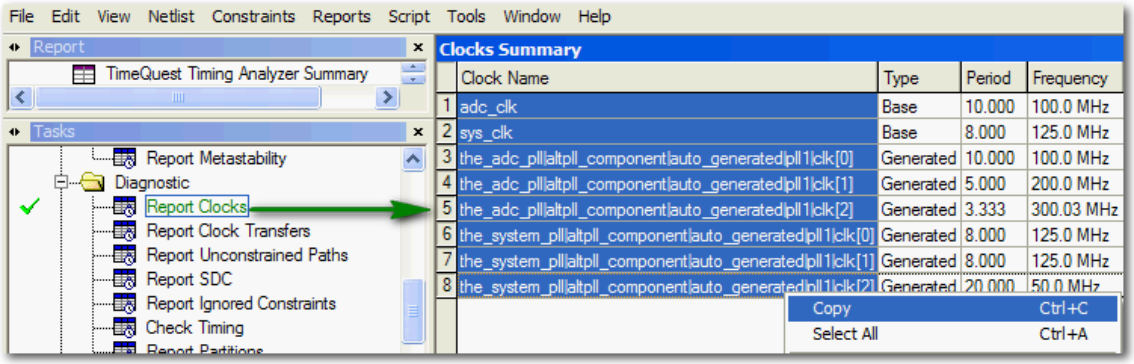
set_false_path -from [get_clocks sys_clk] -to [get_clocks dsp_clk]
set_false_path -from [get_clocks dsp_clk] -to [sys_clk]

This works fine when there are only a few clocks, but quickly grows to a huge number of assignments that are completely unreadable. In a simple design with three PLLs that have multiple outputs, the *set_clock_groups* command can clearly show which clocks are related in less than ten lines, while *set_false_path* may be over 50 lines and be very non-intuitive on what is being cut.

Quick tip for writing set_clock_groups constraint

1) Since derive_pll_clocks is creating many of the clock assignments, the user may not know all of the clock names. A quick way to make this constraint is to create an .sdc with steps 1-3 above, i.e. 1) add *create_clock* on each incoming clock, 2) add *derive_pll_clocks* and 3) add *derive_clock_uncertainty* to your .sdc.

2) Double-click in the left Task panel of TimeQuest on Report Clocks. This will read in your existing SDC and apply it to your design, then report all the clocks. From that report, I highlight all of the names in the first column that I know, right-click copy, as shown below:



You have just copied all the clocks in your design in the exact format TimeQuest recognizes them. Paste them into your .sdc file.

3) Now that you have a columnar list with every clock in the design, format that list into the *set_clock_groups* command. For example, I may start with the following empty example:

```
set_clock_groups -asyncrhonous -group { \
} \
-group { \
} \
-group { \
} \
```

*-group { *
}

And then paste clocks into groups to define how they're related, adding or removing groups as necessary.

4) Finally, format the list of clocks to make it readable. Here is a screenshot of an .sdc:

```
# Core Timing Quick Start Constraints:

create_clock -name sys_clk -period 8.0 [get_ports sys_clk]

create_clock -name adc_clk -period 10.0 [get_ports adc_clk]

derive_pll_clocks

derive_clock_uncertainty


set_clock_groups -asynchronous \
                 -group {adc_clk \
                         the_adc_pll|altpll_component|auto_generated|pll1|clk[0] \
                         the_adc_pll|altpll_component|auto_generated|pll1|clk[1] \
                         the_adc_pll|altpll_component|auto_generated|pll1|clk[2] \
                        } \
                 -group {sys_clk \
                         the_system_pll|altpll_component|auto_generated|pll1|clk[0] \
                         the_system_pll|altpll_component|auto_generated|pll1|clk[1] \
                        } \
                 -group {the_system_pll|altpll_component|auto_generated|pll1|clk[2]}
```

Note that the last group has a PLL output system_pll|..|clk[2] while I put the input clock and other PLL outputs into a different group. That is because I made this clock a frequency that can't be related to the other clocks, and must be treated asynchronously to them. Usually most outputs of a PLL are related and hence in the same group, but it's not a requirement, and up to the user's design.

That's it. For many designs, that is all that's necessary to constrain the core. Some common core constraints that will not be covered in this quick start section that user's do are:

- Add multicycles between registers which can be analyzed at a slower rate than the default analysis, i.e. opening the window. For example, a 10ns clock period will have a 10ns setup relationship. If the data changes at a slower rate, or perhaps the registers toggle at a slower rate due to a clock enable, than the user wants to apply a multicycle that opens the the window that the data passes through. This will be a multiple of the clock period, making the setup relationship 20ns, 40ns, etc., while keeping the hold relationship at 0ns. These types of multicycles are generally applied to paths.

- The second common form of multicycle is when the user wants to shift the window. This generally occurs when the user does a small phase-shift on a clock. For example, if the user has a 10ns clock coming out of a PLL, and second clock coming out that is also 10ns but with a 0.5ns phase-shift, the default setup relationship from the main clock to the phase-shifted clock is 0.5ns and the hold relationship is -9.5ns. It is almost impossible to meet a 0.5ns setup relationship, and most likely the user wants data to transfer in the next window. By adding a multicycle from the main clock to the phase-shifted clock, the setup relationship becomes 10.5ns and the hold relationship becomes 0.5ns. This multicycle is generally applied between clocks

and is something the user should think about as soon as they do a small phase-shift on a clock. This multicycle is called shifting the window.

If any of this discussion on default setup and hold relationships is confusing, please read the basics of setup and hold, as well as the following section on determining default setup and hold relationships.

- Add a *create_generated_clock* to ripple clocks. Basically anytime a register's output drives the .clk port of another register, that is a ripple clock. Clocks do not propagate through registers, so all ripple clocks must have a *create_generated_clock* constraint applied to them for correct analysis. Unconstrained ripple clocks will show up in TimeQuest's task "Report Unconstrained Paths", so they are easily recognized. In general, ripple clocks should be avoided for many reasons, and if possible, a clock enable should be used instead.

- Add a *create_generated_clock* to clock mux outputs. Without this, all clocks propagate through the mux and will be related. TimeQuest will analysis paths downstream from the mux where one clock input feeds the source register and the other clock input feeds the destination, and vice-versa. Although it could be valid, this is usually not what user's want. By putting *create_generated_clock* constraints on the mux output, relating them to the clocks coming into the mux, the user can correctly group these clocks with other clocks.

I/O Timing

(Note: This section does not explicitly cover source-synchronous interfaces, although they use the same principles.)

There are only two I/O specific .sdc commands, *set_input_delay* and *set_output_delay*, and they can be difficult to grasp at first. The most important concept is that these constraints describe what is going on outside of the FPGA, and with that information TimeQuest figures out what is required inside the FPGA. I break this down into 5 steps, which is important for the first time through, although quickly becomes intuitive:

Steps for I/O Timing:

1) Add *create_clock* to create a virtual clock for the I/O interface
2) Add *set_input_delay* or *set_output_delay* to the I/O port/s. Add it twice, once using the option -min and once using -max, and have 0.0 be the value in both cases. (This will be modified in step 5)
3) Determine the default setup and hold relationships between the FPGA clock and the virtual clock
4) Add multicycles if these default relationships are not correct
5) Modify the -max and -min delay values to account for external delays

I want to point out that the values used for the set_input_delay and set_output_delay are entered last, which is the opposite of what most new users do. Going through the first steps will make it apparent why. Also note that bidirectional I/O are really analyzed as inputs and outputs, so they usually have both set_input_delay and set_output_delay assignments.

Step 1) Use *create_clock* to add a virtual clock for the I/O interface

This is always the first step, which is certainly not intuitive. If the FPGA communicates with a PCI device that runs at 66MHz and a DAC running at 200MHz then the user might add the following to their SDC file:

create_clock -period 15.151 -name pci_clk_ext
create_clock -period 5.0 -name dac_clk_ext

Note that I did not apply these clocks to anything in the FPGA, which is what makes them virtual clocks; they exist outside of the FPGA. How this will be used becomes apparent in the next few steps, but this step is usually easy since it reflects what's occurring in hardware.

Step 2) Add *set_input_delay* or *set_output_delay* on the I/O port/s

Add it twice, once using -min and once using -max. Use the value 0.0 for both delays, and the virtual clock for the clock.

The instructions are long, but it's really quite easy. If constraining an output port called DAC_DATA[5], I might put in my .sdc:

set_output_delay -clock dac_clk_ext -max 0.0 [get_ports DAC_DATA[5]]
set_output_delay -clock dac_clk_ext -min 0.0 [get_ports DAC_DATA[5]]

As specified, the -clock option was filled with the virtual clock created in step 1), and the -max and -min values are 0.0. That 0.0 is just a placeholder we will modify in step 5.

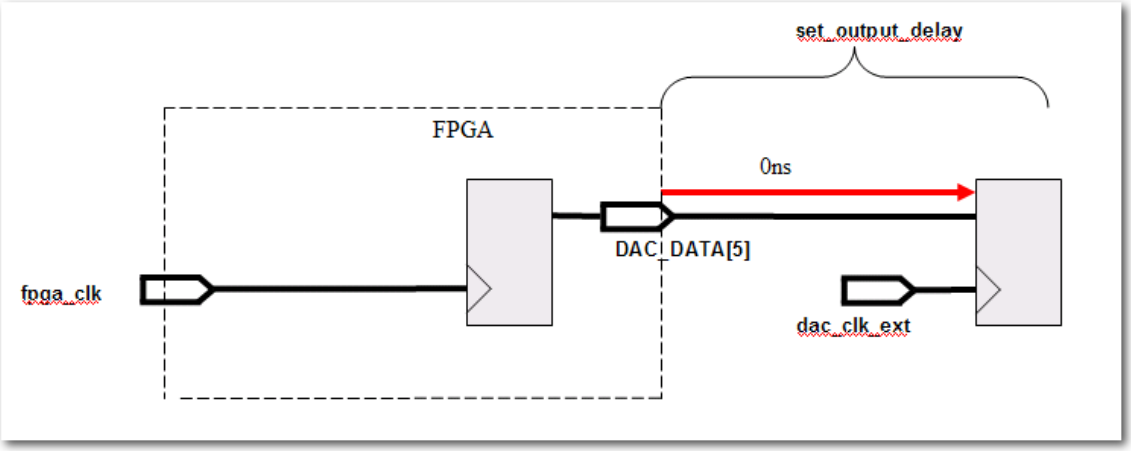
For an input bus where I want all ports to have the same constraint, I might do:

set_input_delay -clock adc_clk_ext -max 0.0 [get_ports ADC_DATA[*]]
set_input_delay -clock adc_clk_ext -min 0.0 [get_ports ADC_DATA[*]]

This step is straightforward since it's just following the instructions without any analysis, but it's important to understand what the command does. Intrinsically they do not really "constrain" anything, and instead describe what is going on outside of the FPGA. Looking at our output constraint, let's break down its components:

1. set_output_delay	There is a register being driven by an FPGA output
2. -clock dac_clk_ext	This register is clocked by our virtual clock dac_clk_ext
3. -max/-min 0.0	The external delay has a max of 0.0 and min of 0.0
4. [get_ports DAC_DATA[5]	The register is driven by port DAC_DATA[5]

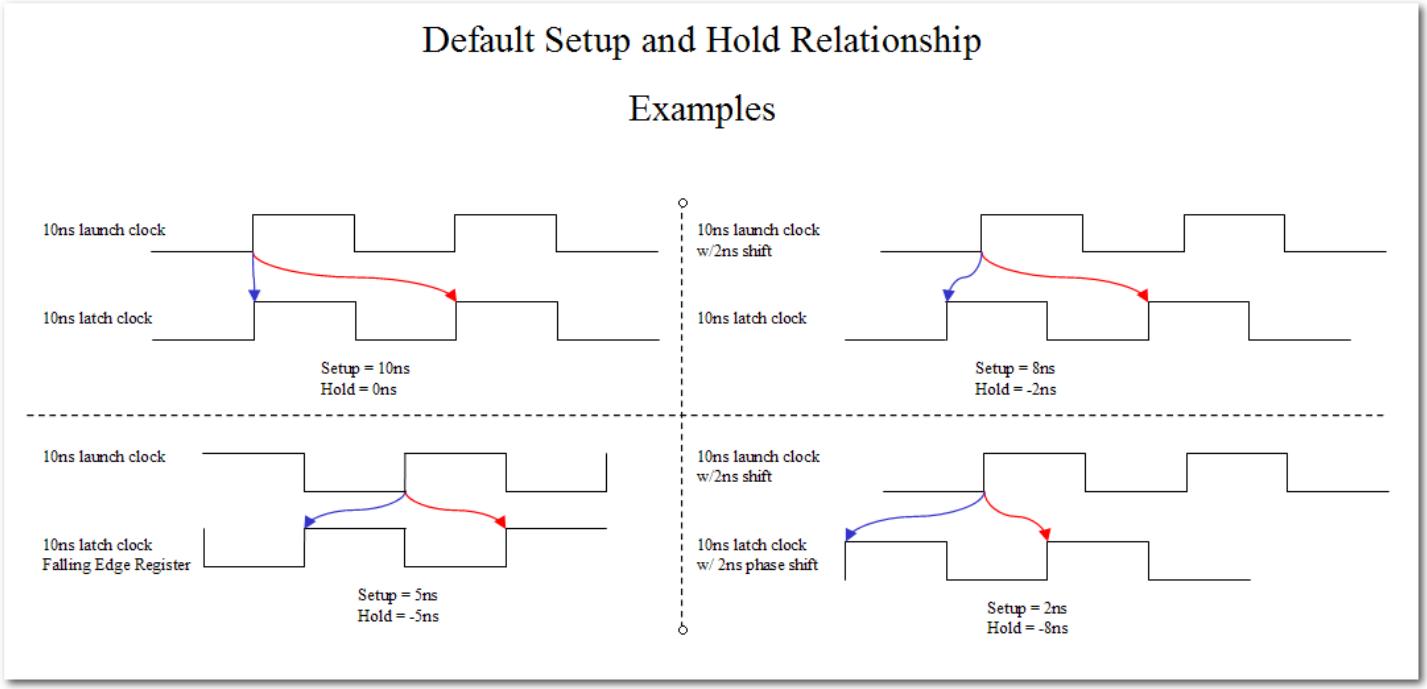
Let's look at this command in schematic format:



As can be seen, we just described a circuit outside the FPGA. We also now have a register feeding another register. This is the standard path analysis done on every path inside the FPGA.

Step 3) Determine the default setup and hold relationship between the FPGA clock and virtual clock

This step requires the user to determine the setup and hold relationship between the clock inside the FPGA and the virtual external clock. This is usually very straightforward, as in most cases the launch clock and latch clock have the same period and are edge aligned, and hence have a setup relationship equal to the clock period, and a hold relationship of 0ns, as shown in the top-left example:

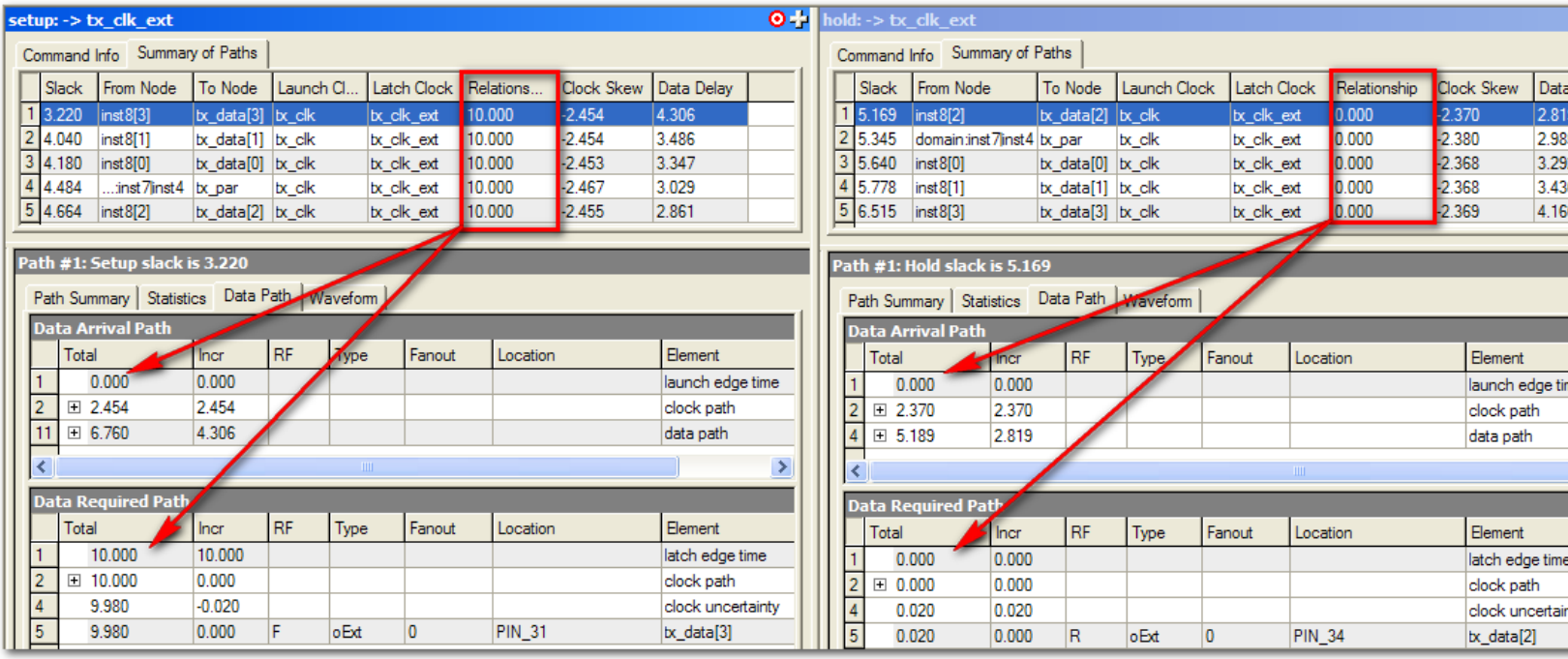


For I/O, the virtual clock will be the launch clock for input constraints, and the latch clock for output constraints. I've shown a few more cases, but won't delve into too much detail on how to determine the setup and hold relationship, which is covered in-depth in Timing Analysis Basics.

Rather than delve into calculating the relationship, I'll show how TimeQuest will show the relationship it is using, and hence the user doesn't have to figure it out beforehand. For example, a user might have a 100MHz clock coming into the FPGA, which goes through a PLL and drives data out at 100Mbps. After following the previous steps, the user creates a 10ns external clock and applies it to the output ports like so:

create_clock -period 10.0 -name tx_clk_ext
set_output_delay -clock tx_clk_ext -max 0.0 [get_ports {TX_DATA[*] TX_PAR}]
set_output_delay -clock tx_clk_ext -min 0.0 [get_ports TX_DATA[*] TX_PAR}]

In this example, they've created a virtual clock called tx_clk_ext. They also said the ports TX_DATA[*] and TX_PAR drive external registers clocked by tx_clk_ext, and the max and min delay to those registers is 0.0ns. Since the internal clock also has a period of 10.0ns, and neither is phase-shifted, then the default setup relationship is 10.0ns and the default hold relationship is 0.0ns. If the user is unsure of this, they can read in the .sdc file using the iterative method and run report_timing to those ports. In TimeQuest's pull-down menu: Reports -> Custom Reports -> Report Timing. Simply put the virtual clock name, *tx_clk_ext*, in the To Clock section, and run report_timing twice, once for setup and once for hold. In this example, I got the following two reports:



The left panel is the setup analysis, and the Relationship column, shown in red, is 10ns. This is expected, as we have a 10ns clock feeding another 10ns clock. Below the Summary of Paths is the detail for the first path. The 10ns is used such that the launch edge time is 0ns and

the latch edge time is 10ns. Likewise for hold analysis, the relationship is 0ns. The launch edge time is 0ns and the latch edge time is 0ns.

So what is this really saying? When the launch clock comes into the FPGA, travels to the source register, which is the output register in this case, then through the output port to the external register, it must get there in greater than 0ns(the hold relationship) and less than 10ns(the setup relationship). Since our external -max and -min delays are 0ns, i.e. there is no external delay, than the delay within the FPGA must be greater than 0ns and less than 10ns. At this point, we have a full constraint that TimeQuest can analyze, but it is probably not the analysis we want.

Step 4) Add multicycles

This step is usually unnecessary. But if step 3) resulted in a default analysis that is incorrect, the user may want to modify the setup and hold relationships with multicycles. The most common cases for this are when the user wants to open the window or shift the window. Note that we are not accounting for external delays like the Tsu or Tco of an external device or board delays, as that will be done in step 5. This step is just to make sure the clock relationships are correct.

An example would be interfacing to a flash device that takes multiple clock cycles to perform each operation, than the user may want to open the window. An example may look like so:

set_multicycle_path -setup -to [get_ports {FLASH_DATA[*]}] 4
set_multicycle_path -hold -to [get_ports {FLASH_DATA[*]}] 3

These two assignments tell TimeQuest that there are 4 clock cycles for the FLASH_DATA to get out of the FPGA. So if the original setup and hold relationships were 10ns and 0ns, they would now be 40ns and 0ns(assuming the clock period is 10ns).

If the clock inside the FPGA has a phase-shift, generally through a PLL, and the external clock does not, then the user may want to shift the window. For example, if the FPGA clock feeding an output register were phase-shifted -500ps in order to help meet output timing, the default setup relationship would be 500ps. To shift the window, the user would add:

set_multicycle_path -setup -to [get_ports {FLASH_DATA[*]}] 2

or:

set_multicycle_path -setup -from [get_clocks {pll|clk[0]} -to [get_clocks clk_ext] 2

The first one modifies the clock relationship on the output path, while the second one modifies all relationships between these clocks. Either one of these will work as long as they cover what the user wants covered. If the clocks have a 10ns period, the multicycle will modify the setup relationship from 0.5ns to 9.5ns, and the hold relationship from -9.5ns to 0.5ns.

Note that if the FPGA clock were phase-shifted forward a little, then a multicycle would most likely not be necessary. If the default setup relationship were 10ns, and the source clock inside the FPGA were phase-shifted 500ps forward, then the default relationship would become 9.5ns, which is probably what the user wants.

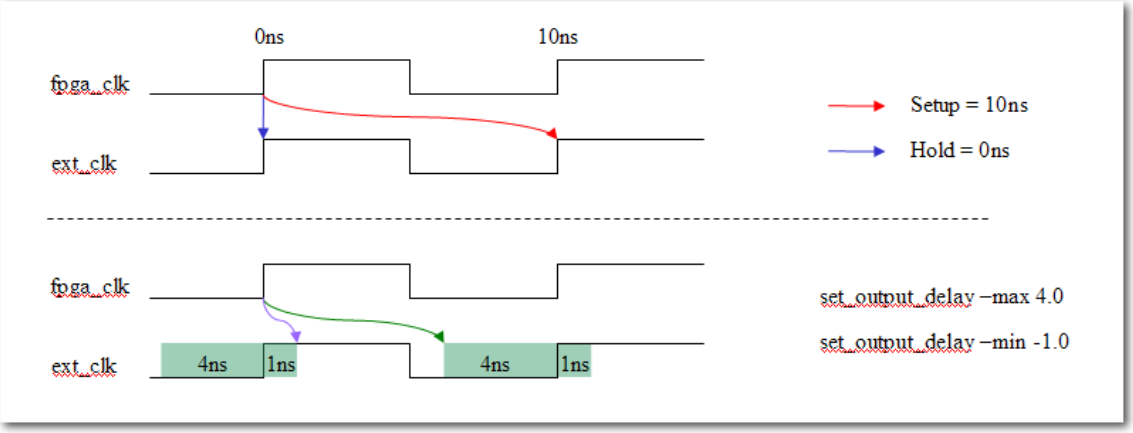
Inputs work the opposite, whereby if the user phase-shifts the latching clock inside the FPGA forward a little, then they probably want a multicycle to shift the window, but if they phase-shift the clock back, they probably do not. This is easily seen by drawing the launch and latch clock waveforms, as explained in the section on default setup and hold, specifically the affect of phase-shifts.

Step 5) Modify the -max and -min delays to account for external delays.

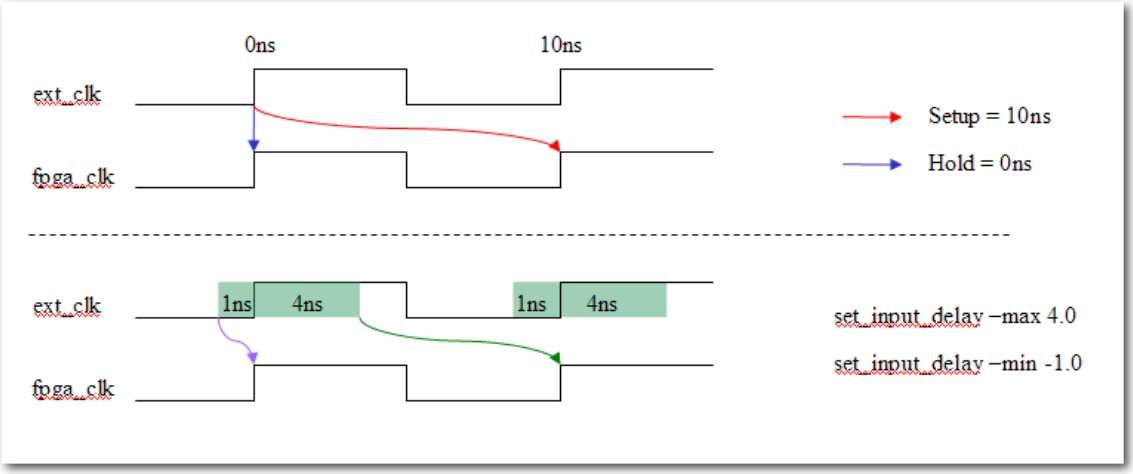
Now that we have the correct setup and hold relationship, it is time to modify the -max and -min values. They are currently set to 0, which means there is no external delay to the register. This allows the entire data window to be used by FPGA delays. In reality, part of the data window is used by the external device and board delays, only leaving part of the data window for the FPGA. The -max and -min values account for these external delays. Let's see how they affect the analysis before determining how they account for external delays.

With both -max and -min at 0, we are stating that there are no external delays, and basically have no affect on the analysis. As the -max value gets larger, it cuts into our setup relationship. So if the our default setup relationship were 10ns, and the -max output delay were 4ns, that would mean the FPGA must get it's data out in less than 6ns to meet timing. Note the setup relationship is still 10ns, but the FPGA's delay plus the external delay must be less than that. So the larger -max gets, the more quickly the FPGA needs to get its data out and the harder it is to meet timing. As the -max value gets larger, the FPGA needs a faster Tco.

The -min value is often more confusing because it works in the opposite way, whereby the smaller it gets, the harder it is to meet timing. If the hold relationship is 0ns, and the -min value was -1ns, then the only way to meet timing would be for the FPGA to get its data out in more than 1ns. Looking at this through waveforms:



The top waveform shows the default relationships, while the second waveform shows what happens after accounting for the external delays. Rather than the FPGA needing to get it's data out between 0ns and 10ns, it must now get out between 1ns and 6ns. Likewise, for internal paths, if a user entered similar external delays:



It may seem confusing that the green and purple arrows do not start at the same point. What's being shown is that when the external clock launches data, it can take anywhere from -1ns to +4ns to reach the FPGA. We use the larger number for the setup analysis, and the smaller number for the hold analysis.

How these external delays are actually added into the timing reports is shown in the upcoming section on correlating constraints to the timing reports.

One thing to note is that, as the difference between -max and -min values grows, the more difficult it is for the FPGA to meet timing. In the output example above, the default relationship says the data must transfer between 0ns and 10ns. As the external delay spreads from our original placeholder of 0ns for -min and -max, to -1ns and 4ns, the external device now uses 5ns of that 10ns window, and so the FPGA only has 5ns to work with. I wanted to point this out, because users often don't see the relationship right away, and it often helps with understanding.

So now that we conceptually know how the external delays work, let's account for real external delays by looking at the output side first:

External device parameters:
- Tsu_ext = Tsu of external device
- Th_ext = Th of external device

Data delays on board:
- Max_fpga2ext = Max board delay to external device
- min_fpga2ext = min board delay to external device

set_output_delay -max = Tsu_ext + Max_fpga2ext
set_output_delay -min = -Th_ext + Min_fpga2ext

For input constraints, they look like so:

External device parameters:
- Tco_ext = Tsu of external device
- minTco_ext = Th of external device

Data delays on board:
- Max_ext2fpga = Max board delay from external device to FPGA
- min_ext2fpga = min board delay from external device to FPGA

Clock delays on board:

Max_clk2fpga = Max delay from board clock to FPGA
min_clk2fpga = min board delay from clock to FPGA
Max_clk2ext = Max delay from board clock to external device
min_clk2ext = min board delay from clock to external device

set_input_delay -max = Tco_ext + Max_ext2fpg
set_input_delay -min = minTco_ext + min_ext2fpga

The user could actually put variables and equations into their .sdc file, which is shown in the Tcl Syntax section.

Note that these equations did not take into account board level clock skew, and is basically assuming the clock to the FPGA and external device are equivalent. There is a very nice .sdc constraint for entering board-level clock delays, which I will show in a second, but what I see most users do is roll their board-level clock skews into the -max and -min values. (Note that clock skew is positive when the delay to the destination is larger than the delay to the source). Anyway, rolling clock skew into the delays looks like so:

External device parameters:
Tsu_ext = Tsu of external device
Th_ext = Th of external device
Data delays on board:
Max_fpga2ext = Max board delay to external device
min_fpga2ext = min board delay to external device
Clock delays on board:
Max_clk2fpga = Max delay from board clock to FPGA
min_clk2ext = min board delay from clock to external device
Max_clk2ext = Max delay from board clock to external device
min_clk2fpga = min board delay from clock to FPGA

set_output_delay -max = Tsu_ext + Max_fpga2ext - (min_clk2ext - Max_clk2fpga)
= Tsu_ext + Max_fpga2ext - (min_clk_skew)
set_output_delay -min = -Th_ext + min_fpga2ext - (Max_clk2ext - min_clk2fpga)
= -Th_ext + min_fpga2ext - (Max_clk_skew)

For input constraints:

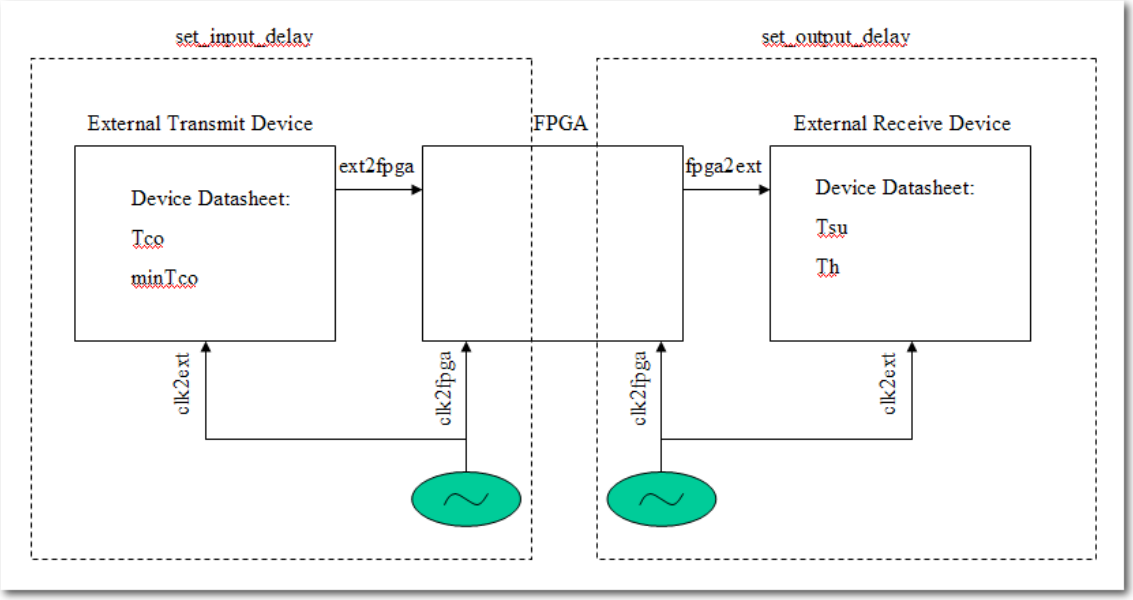
External device parameters:
Tco_ext = Tco of external device
minTco_ext = min Tco of external device
Data delays on board:
Max_ext2fpga = Max board delay from external device to FPGA
min_ext2fpga = min board delay from external device to FPGA
Clock delays on board:
Max_clk2fpga = Max delay from board clock to FPGA
min_clk2fpga = min board delay from clock to FPGA
Max_clk2ext = Max delay from board clock to external device
min_clk2ext = min board delay from clock to external device

set_input_delay -max = Tco_ext + Max_ext2fpg - (min_clk2fpga - Max_clk2ext)
= Tco_ext + Max_ext2fpg - (min_clk_skew)

set_input_delay -min = minTco_ext + min_ext2fpga - (Max_clk2fpga - min_clk2ext)

= minTco_ext + min_ext2fpga - (Max_clk_skew)

Here's a diagram:



The on-board clock source is shown twice, once for the input and once for the output, but often they are the same source.

Again, the equations are given above, and I find most people roll their clock delays into the FPGA -max and -min values. That being said, SDC has a very nice constraint that allows the user to enter board-level clock delays externally:

set_clock_latency -source -late 2.0 [get_clocks clk_fpga]
set_clock_latency -source -early 1.8 [get_clocks clk_fpga]
set_clock_latency -source -late 2.3 [get_clocks clk_ext]
set_clock_latency -source -early 2.1 [get_clocks clk_ext]

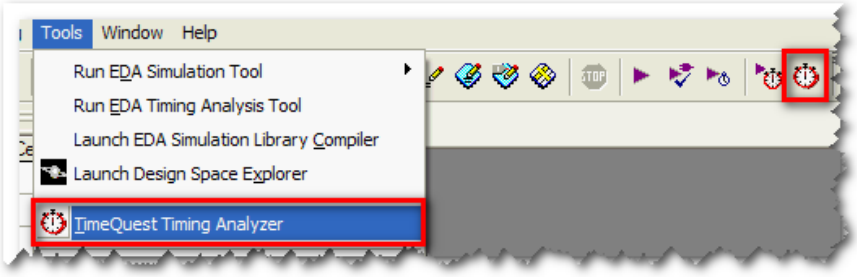
TimeQuest will then properly roll these into the timing analysis. This is very nice in that it simplifies worrying about clock skew, what sign to use and whether to add or subtract delays, as the analysis takes care of it all for you. Whether you want to roll board-level clock delays into the external -max/-min delays, or use set_clock_latency, is purely up to the user's preference. If done correctly, the analysis will be the same either way.

Analyzing Results

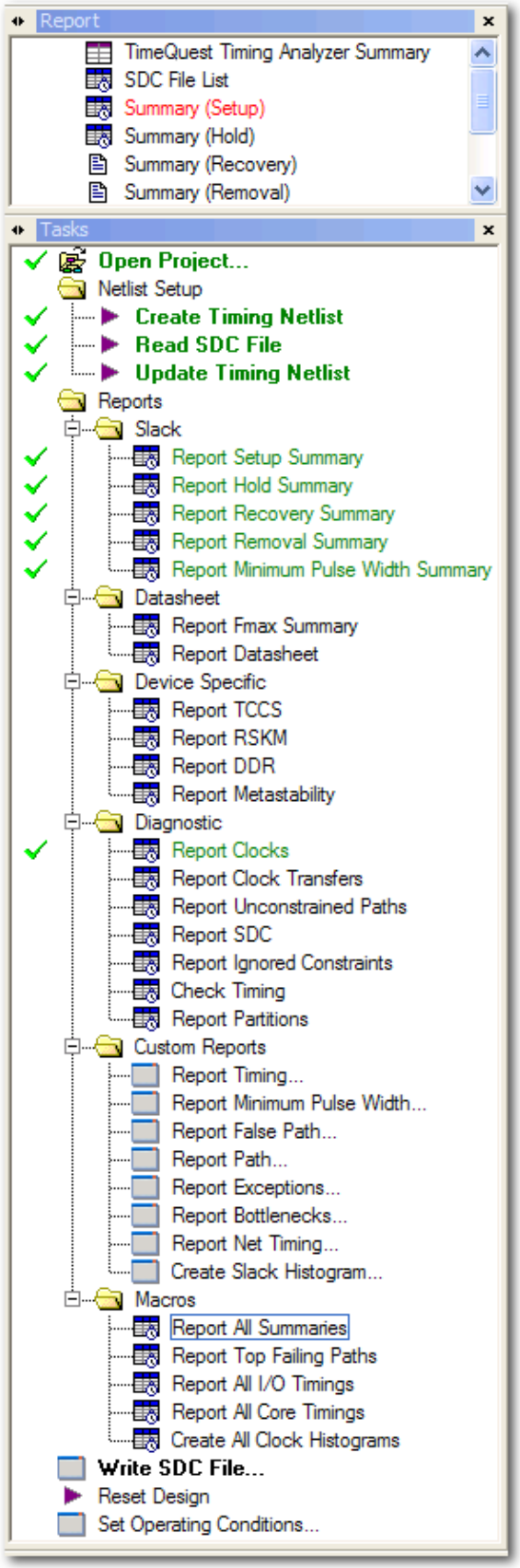
This is one of the most important sections for getting started, not because it's overly difficult, but because most other documents gloss over the analysis. I see time and time again where users concentrate on their .sdc files without understanding what it will look like in the final analysis. Knowing what your constraints will look like when analyzing a path is one of the most important skills, since it completes the user's understanding and lets them correlate their .sdc input to the back-end analysis, and from their determine how the FPGA delay's affect timing.

The Iterative Methodology

When entering constraints, users will make mistakes, and want a quick method to modify their .sdc files, analyze the results, then repeat. First, launch TimeQuest, either from the Tools pull-down menu or the TimeQuest button:



Once open, the first thing I would recommend clicking on is the Task's Macro "Report all Summaries", shown below:



Doing so will run the three steps in Netlist Setup. These are:

1) Create Timing Netlist. The default is to create a slow timing model netlist. If the user wants a different netlist, they should access Create Timing Netlist from the Netlist pull-down menu.

2) Read SDC file will read in the user's SDC files. If any were added in Quartus II's Assignment -> Settings -> TimeQuest or -> Files, they will be read in, otherwise TimeQuest will look for any .sdc files matching the project name. If the user makes changes to the TimeQuest .sdc list in Quartus II, they must re-launch TimeQuest for those changes to take affect.

3) Update Timing Netlist will then apply the SDC constraints to the design netlist.

4) The Report All Summaries macro will the run Setup, Hold, Recovery, Removal Summaries, as well as Minimum Pulse Width checks. This is basically a summary analysis of every constrained path in the design. (Device Specific checks are not run…)

The iterative method is when the user makes a change to their .sdc. I recommend user's edit .sdc files from within TimeQuest or Quartus II. Besides syntax coloring, there are pop-ups to assist command syntax, as well as the power of entering constraints with a GUI using the SDC editor's Edit -> Insert Constraint.

Once a user modifies their .sdc file and saves it, they should double-click Reset Design. This takes TimeQuest back to the point where it has created the timing netlist but not yet read in the .sdc files. Double-clicking Report All Summaries will re-read in the edited .sdc files and re-create the timing summaries.

In essence, the iterative method is:

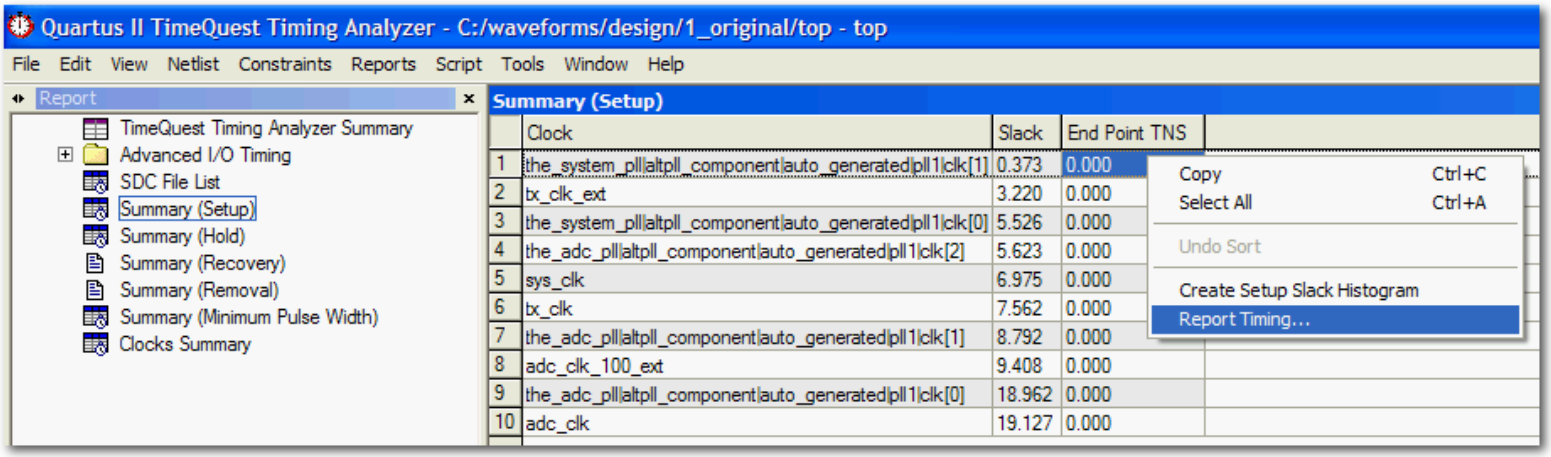
1) Open TimeQuest
2) Double-click "Report All Summaries"
3) Analyze results
4) Make changes to .sdc file and save
5) Double-click "Reset Design"
6) Double-click “Report All Summaries”
7) Analyze results
8) Repeat steps 4-7 as necessary.

Be aware that this method just re-runs timing analysis using new constraints, but the fit being analyzed has not change. The place-and-route was run with the old constraints, but the user is analyzing with new constraints, so if something is failing timing against these new constraints, it may just be that the user needs to run place-and-route again.

For example, the fitter may concentrate on a very long path in the user’s design, trying to close timing. Within TimeQuest, the user may realize this path runs at a lower rate, and so they add *set_multicycle_path* assignments to open the window. Running TimeQuest iteratively with these new multicycles, those paths no longer show up but something else does. The paths may have sub-optimal placement since the fitter was concentrating on the other paths when it ran, since they were more critical. The iterative method is recommend for getting the .sdc files correct, but the user will have to re-run a full compile to see what Quartus II can do with those constraints.

A diving tool

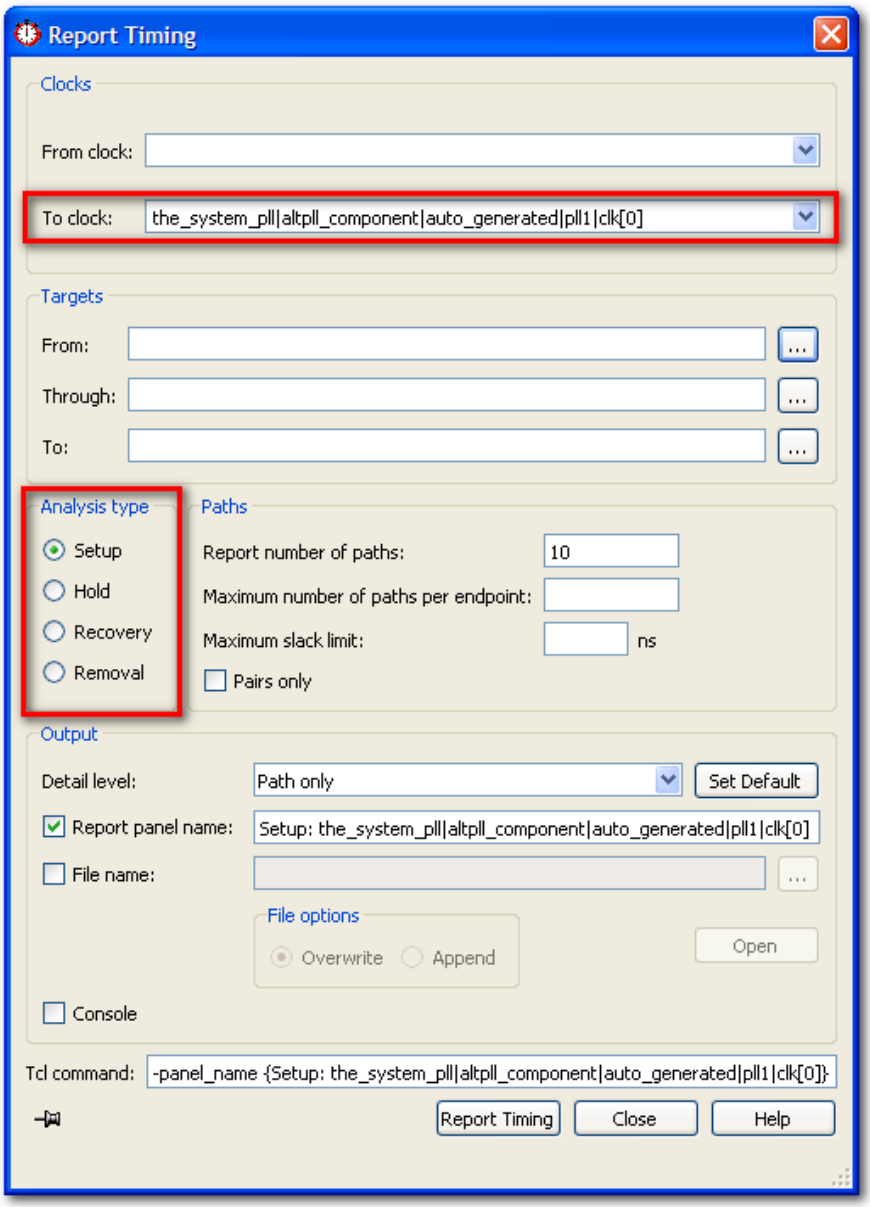
The previous section had users run "Report All Summaries". This will run the four major types of analysis on every constrained clock domain in the design: setup, hold, recovery and removal. The top-left TimeQuest box is called Reports, and is similar to a table of contents for all the reports created. Highlighting any name in the Report's box will show that report in the main viewing pane. Below is a design with the Summary (Setup) report highlighted:



The main viewing pane shows the Slack for every clock domain. For example, row 5 says that, for every path where sys_clk feeds the destination register, the worst slack is 6.975ns.

Positive slack is good, saying these paths meet timing by that much. The End Point TNS stands for Total Negative Slack, and is the sum of all slacks for each destination and can be used as a relative assessment of how much a domain is failing.

Of course, this is just a summary. To get details on any domain, the user should right-click that row and select Report Timing…



The report_timing dialogue box appears, auto-filled with the Setup radio button selected and the To Clock filled with the selected clock. This is done because the user was looking in the Setup Summary report, and right-clicked on that particular clock. As such, the worst 10 paths where that is the destination clock will be reported. The user can modify the settings any way they want, such as increasing the number of paths to report, adding a Target filter, adding a From Clock, writing the report to a text file, etc.

Note that any report_timing command can be copied from the console at the bottom into a user-created Tcl file, so that a user can analyze specific paths again in the future without having to click so many buttons. This is often done as users become more comfortable with TimeQuest and find themselves analyzing the same problematic parts of their design over and over, but is by no means required. Many complex designs successfully use TimeQuest as a diving tool, i.e. just starting with summaries and diving down into the failing paths after each compile.

report_timing

The command *report_timing* is by far the most important analysis tool in TimeQuest. Many designs require nothing but this command. Because of this, I recommend the user typing "report_timing -long_help" in the TimeQuest console, just to see every option available. This command can be accessed from the Tasks menu on the left, from the pull-down menu Reports -> Custom Reports pull-down menu, and by right-clicking on just about anything in TimeQuest.

Looking at the screen-shot above, the major options are shown for report_timing. The From Clock and To Clock filter paths where the selected clock is used as the launch or latch. The pull-down menu allows you to choose from existing clocks(although admittedly has a "limited view" for long clock names).

The Targets for From and To allow the user to report paths with only particular endpoints. These are usually filled with register names or I/O ports, and can be wildcarded. For example, a user might do the following to only report paths within a hierarchy of interest:

*report_timing -from *|egress:egress_inst|* -to *|egress:egress_inst|* -(other options)*

If the -from/-to/-through options are empty, then it is assumed to be *, i.e. all possible targets in the device. The -through option is to limit the report for paths that pass through combinatorial logic, or a particular pin on a cell. My experience is this is seldom used, and troublesome to rely on due to combinatorial node name changes during synthesis. I try to only use -from and -to options when possible. Also, the […] box after each target will open the Name Finder, which is a GUI for searching on specific names. This is especially useful to make sure the name being entered matches nodes in the design, since the Name Finder can immediately show what matches a user's wildcard.

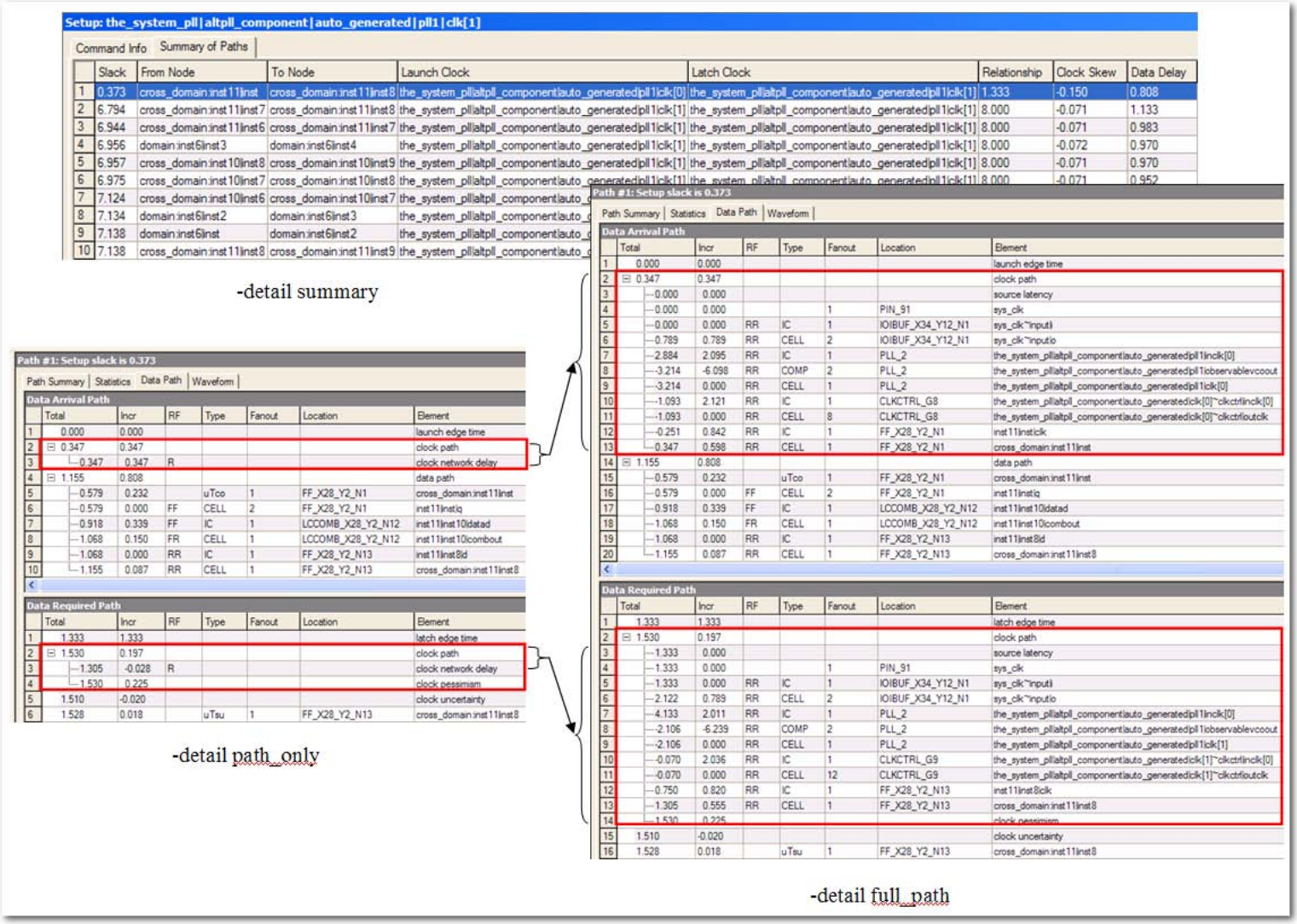
The Analysis type will be -setup, -hold, -recovery or -removal. These will be explained in more detail later, as understanding them is the underpinning of timing analysis.

The Detail level, -detail, is an option often glanced over that should be understood. It has four options, but I will only discuss three. The first level is called Summary, and will only give Summary information, specifically the Source Register, Destination Register, Source Clock, Destination Clock, Slack, Setup Relationship, Clock Skew and Data Delay. The summary report is **always** reported with more detailed reports, so the user would choose this if they want less info. A good use for summary detail is when writing the report to a text file, where *-detail summary* can be quite brief.

The next level is *-detail path_only*. This report gives all the detailed information, except the Data Path tab will show the clock tree as one line item. This is useful when the user knows the clock tree is correct, and does not want to be bothered with all the details. This is common for most paths within the FPGA. A useful data point is to look at the Clock Skew column in the summary report(which is shown for all options of -detail), and if it's a small number, say less than +/-150ps, then the clock tree is well balanced between source and destination.

If there is clock skew, the detail option should be set to *-detail full_path*. This breaks the clock tree out into explicit detail, showing every cell it goes through, including such things as the input buffer, PLL, global buffer(called CLKCTRL_), and any logic. If there is clock skew, this is the way the user determines what in their design is causing the clock skew. The -detail full_path option is also recommended for I/O analysis, since only the source clock or destination clock is inside the FPGA, and therefore its delay plays a critical role in meeting timing.

Here are screen-shots of the same path analyzed with *-detail summary*, *-detail path_only*, and *-detail full_path*. Note that the clock delays are identical between path_only and full_path, but full_path has more details:



Quartus II 9.1 added the +/- feature to the Data Path report, whereby a user can "roll-up" their clock and data path. So *-detail full_path* can be used all the time, and the user would roll-up the clock tree if they don't need it. The *-detail path_only* is more useful is when writing to a text file, which does not have the roll-up feature, or when locating a path to the Chip Planner, so it does not also locate the clock path.

The Data Path tab of a detailed report gives the delay break-downs, but there is also useful information in the Path Summary and Statistics tabs, while the Waveform tab is useful to help visualize the Data Path analysis. I would suggest taking a few minutes to look at these in the user's design. The whole analysis takes some time to get comfortable with, but hopefully is clear in what it's doing.

Report_timing also has the Panel Name, which is what name will be used in TimeQuest's Report section. There is also an optional -file, which allows the user to write the information to a file. If they name the file <filename>.htm, it will write out an HTML report.

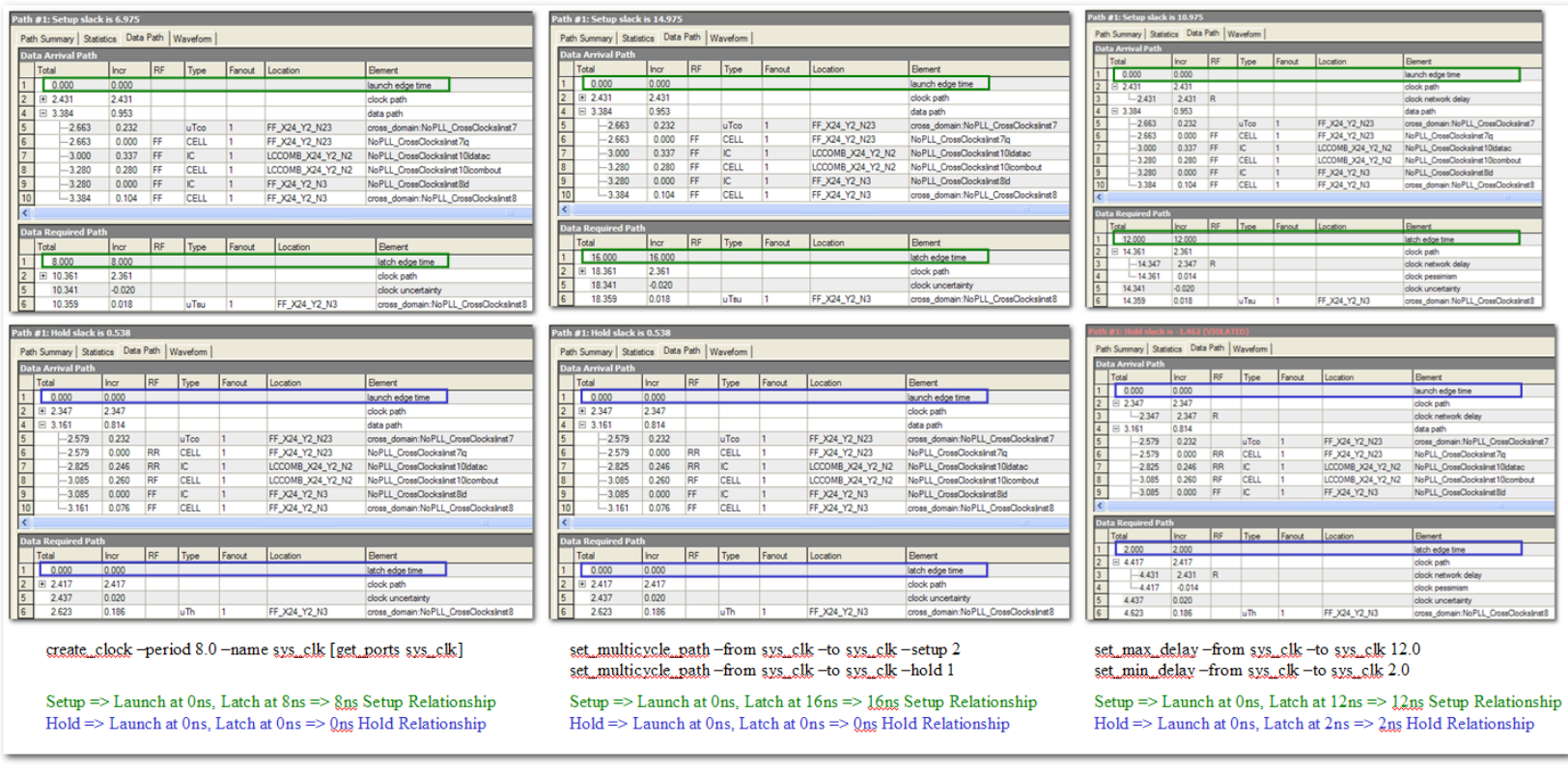
The command *report_timing* shows every path. Two endpoints that have a lot of combinatorial logic between them might have many different paths. Likewise, a single destination may have hundreds of paths leading to it. Because of this, the user might list hundreds of paths, many of which have the same destination and might have the same source.

The checkbox option pairs only, will only list one path for each pair of source and destination. An even more powerful way to filter the report is limit the Maximum Number of Enpoints per Destination. I often set this to 1 and re-run timing analysis.

Finally, at the bottom is the Tcl Command, which shows the Tcl syntax of what is run in TimeQuest. This can be directly edited before running the command. One thing I commonly add is the *-false_path*. With this option, only false paths will be listed. A false path is any path where the launch and latch clock have been defined, but the path was cut with either a *set_false_path* assignment or *set_clock_groups_assignment*. Paths where the launch or latch clock was never constrained are not considered false paths. This command is useful to see if a false path assignment worked and what paths it covers, or to look for paths between clock domains that should not exist. Note that the Task window's Report False Path is nothing more than *report_timing* with the -false_path flag enabled.

Correlating Constraints to the Timing Report

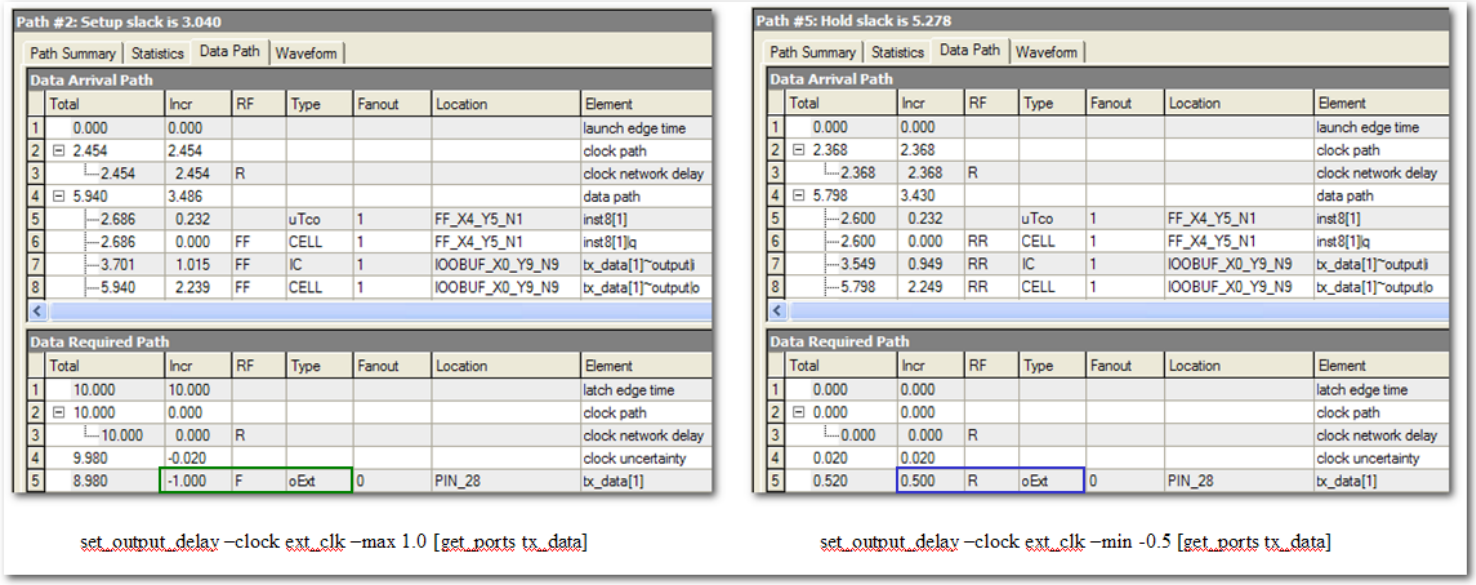
One skill that is seldom explained is how timing constraints show up in the *report_timing* analysis. Most constraints only affect the launch and latch edges. Specifically, *create_clock* and *create_generated_clock* create clocks with default relationships. The command *set_multicycle_path* will modify those default relationships, while *set_max_delay* and *set_min_delay* are low-level overrides that explicitly tell TimeQuest what the launch and latch edges should be. Let's look at the report_timing on a particular path. The top row is setup analsys, and the bottom row is hold analysis.



This is an eyeful, but going from left to right, we start with a clock driving the source and destination registers with a period of 8ns. That gives us a setup relationship of 8ns(launch edge = 0ns, latch edge = 8ns) and hold relationship of 0ns(launch edge = 0ns, latch edge = 0ns). In the

middle column, we have added multicycles to open the window, making the setup relationship 16ns while the hold relationship is still 0ns. In the third column, we use *set_max_delay* and *set_min_delay* constraints to explicitly override the relationships. Note that the only thing changing for these different constraints is the Launch Edge Time and Latch Edge Times for setup and hold analysis. Every other line item comes from delays inside the FPGA and are static for a given fit. Whenever analyzing how the user's constraints affect the timing requirements, this is the place to look.

For I/O, this all holds true except we must add in the -max and -min values. They will be shown as Type iExt or oExt. Let's look at an output port with a *set_output_delay -max 1.0* and *set_output_delay -min -0.5*:



Path #2: Setup slack is 3.040

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 2.454	2.454					clock path
3	└ 2.454	2.454	R				clock network delay
4	⊟ 5.940	3.486					data path
5	├ 2.686	0.232		uTco	1	FF_X4_Y5_N1	inst8[1]
6	├ 2.686	0.000	FF	CELL	1	FF_X4_Y5_N1	inst8[1]|q
7	├ 3.701	1.015	FF	IC	1	IOOBUF_X0_Y9_N9	tx_data[1]~output|i
8	└ 5.940	2.239	FF	CELL	1	IOOBUF_X0_Y9_N9	tx_data[1]~output|o

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	10.000	10.000					latch edge time
2	⊟ 10.000	0.000					clock path
3	└ 10.000	0.000	R				clock network delay
4	9.980	-0.020					clock uncertainty
5	8.980	-1.000	F	oExt	0	PIN_28	tx_data[1]

set_output_delay –clock ext_clk –max 1.0 [get_ports tx_data]

Path #5: Hold slack is 5.278

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 2.368	2.368					clock path
3	└ 2.368	2.368	R				clock network delay
4	⊟ 5.798	3.430					data path
5	├ 2.600	0.232		uTco	1	FF_X4_Y5_N1	inst8[1]
6	├ 2.600	0.000	RR	CELL	1	FF_X4_Y5_N1	inst8[1]|q
7	├ 3.549	0.949	RR	IC	1	IOOBUF_X0_Y9_N9	tx_data[1]~output|i
8	└ 5.798	2.249	RR	CELL	1	IOOBUF_X0_Y9_N9	tx_data[1]~output|o

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					latch edge time
2	⊟ 0.000	0.000					clock path
3	└ 0.000	0.000	R				clock network delay
4	0.020	0.020					clock uncertainty
5	0.520	0.500	R	oExt	0	PIN_28	tx_data[1]

set_output_delay –clock ext_clk –min -0.5 [get_ports tx_data]

Once again, the launch and latch edge times are determined by the clock relationships, multicycles and possibly set_max/min_delay constraints. The set_output_delay's value is also added in as an oExt value. For outputs this value is part of the Data Required Path, since this is the external part of the analysis. The setup report on the left will subtract the -max value, making the setup relationship harder to meet, since we want the Data Arrival Path to be shorter than the Data Required Path. The -min value is also subtracted, which is why a negative number makes hold timing more restrictive, since we want the Data Arrival Path to be longer than the Data Required Path.

Section 2: Timing Analysis Basics

Basics of Setup, Hold, Recovery and Removal

When just learning digital design, usually in school, most designers learn about the setup and hold of a register, calling them Tsu and Th. The Tsu is how long the data must be stable before the clock edge and Th is how long it must be stable after the clock edge. If we violate those requirements, the register can go metastable. These values are a characteristic of the register, and are independent of the clock rates, the place-and-route of the FPGA, etc. We call them micro-parameters, and when used, will reference them as the micro-setup and micro-hold, or µTsu and µTh, of the register. These micro parameters are used by TimeQuest during timing analysis, but they are NOT the fundamentals when we talk about setup and hold relationships. For the most part, the user can ignore these micro parameters since they are always properly calculated by TimeQuest, and should worry about the Setup Relationship and Hold Relationship.

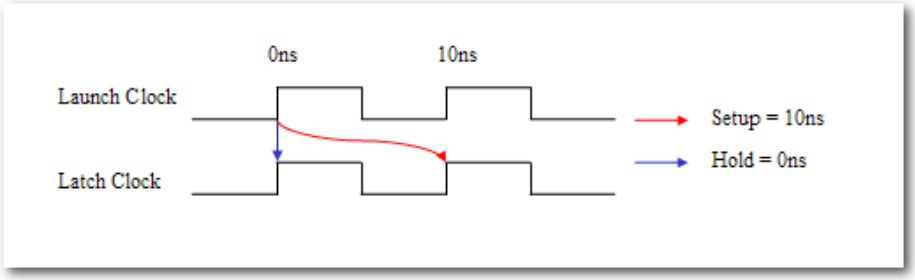
The basic infrastructure of TimeQuest is based on clocks. Clock are first created and applied to the design. Those clocks have relationships within their domain and to other domains. Those relationships create a setup relationship and a hold relationship based on the clocks. These relationships are the fundamental building block of static timing analysis.

Two quick notes before continuing:

Note 1: TimeQuest uses the terms Setup Relationship and Hold Relationship. I will try to follow that nomenclature, but have always thought of them as requirements, and so may say Setup Requirement or Hold Requirement, in which case I mean the same thing. The setup and hold relationships are requirements for the fitter to meet, and are used to determine final timing sign-off, so the two can be inter-changed.

Note 2: Whenever I refer to the Setup Relationship, I also mean the Recovery Relationship. Any mention of the Hold Relationship also includes Removal Relationship. Recovery and Removal are analogous to Setup and Hold, except they deal with signals driving the asynchronous ports on the latching register. This is all discussed in the upcoming Recovery and Removal section. For brevity, I will just write out setup and hold relationship, while inferring recovery and removal.

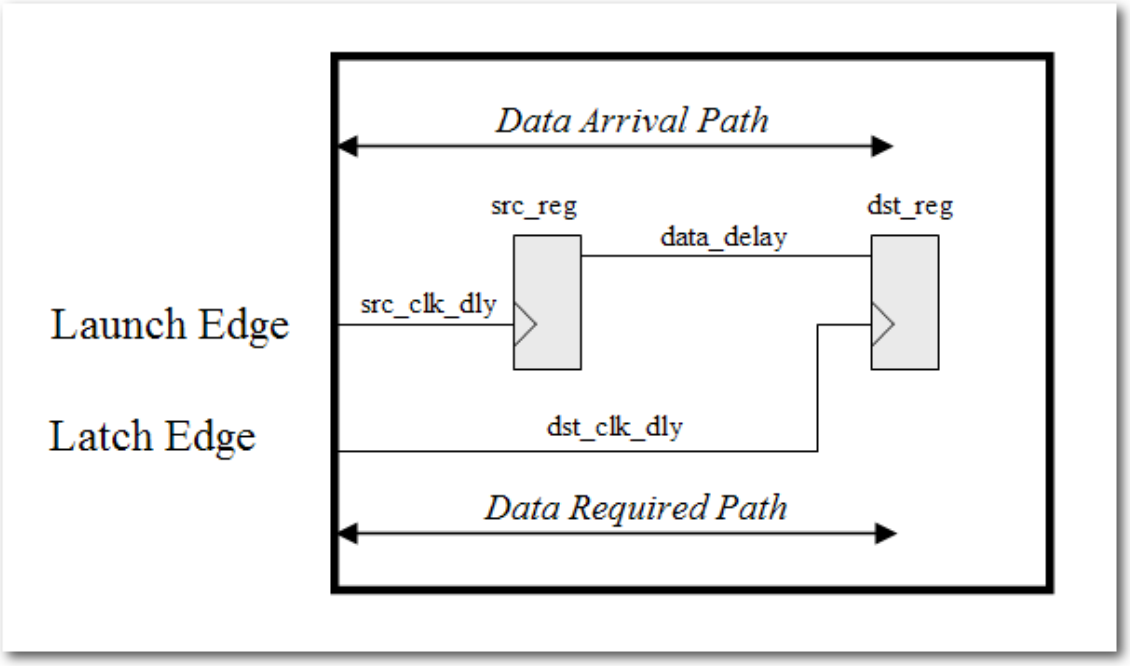
TimeQuest, and static timing analysis for that matter, is based on the principle of repeatable, periodic data relationships. In other words, they rely on clocks. Pretty much every analysis begins with a launch clock and a latch clock. Let's look at a basic case:



This waveform is the fundamental case most users understand without even thinking about it. The setup relationship is 10ns and the hold relationship is 0ns. The setup relationship

means that when the Launch Clock sends a rising edge, it must get to the latch register before the Latch Clock's 10ns edge gets there. The hold relationship means it must get there after the Latch Clock's 0ns edge gets there. Note that this waveform is based on how the .sdc describes the clocks. The Launch clock and Latch clock may be from a *create_clock* or *create_generated_clock* statement, they may be the same clock or different clocks. The relationships are the same regardless, as the Launch and Latch clocks are 10ns clocks, with rising edges at 0ns and falling edges at 5ns.

Let's look at how this applies to a schematic:



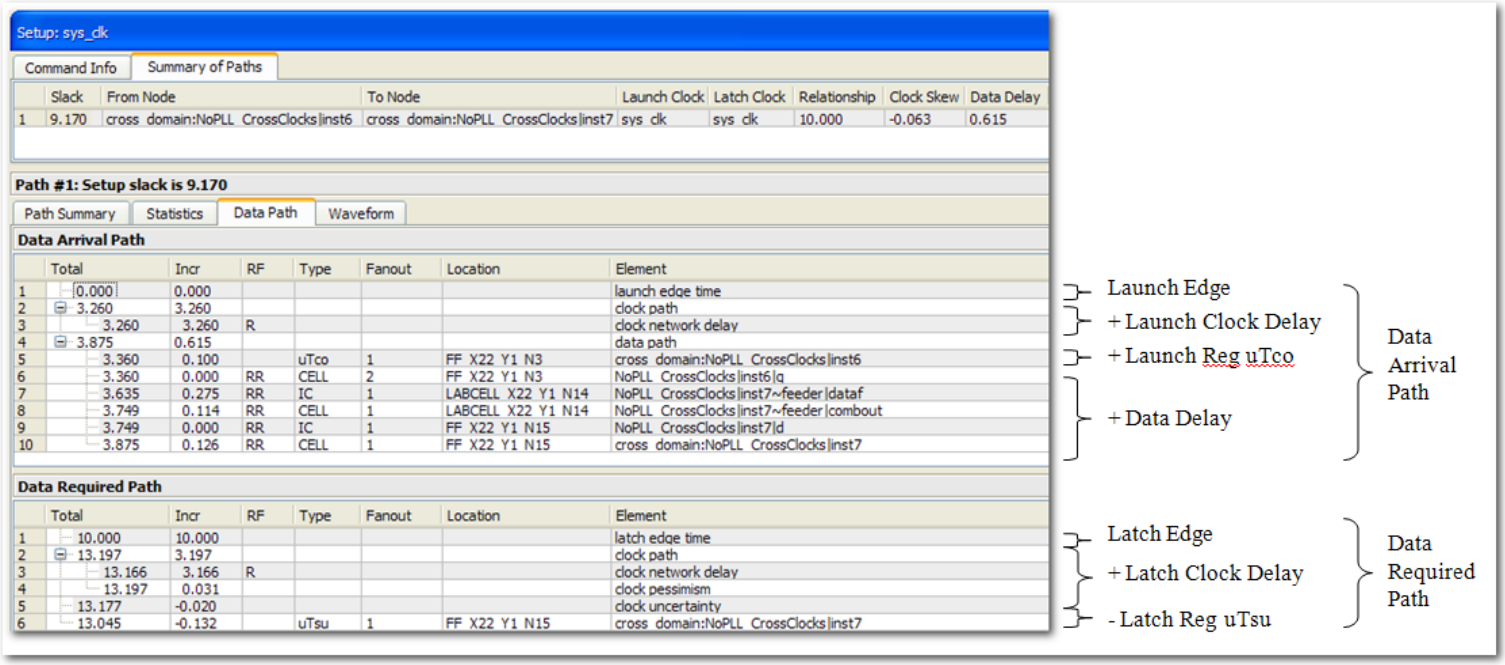
So let's look at this in equation form:

Data Arrival Path = Launch Edge + src_clk_dly + src_reg_uTco + data_delay
Data Required Path = Latch Edge + dst_clk_dly

So when we do a setup check on this path, the Data Arrival Path must get to the FPGA before the Data Required Path's micro setup time(uTsu).

Data Arrival Path + uTsu < Data Required Path
Launch Edge + src_clk_dly + src_reg_uTco + data_delay <
Latch Edge + dst_clk_dly - dst_reg_uTsu

Let's look at this in an actual timing report:

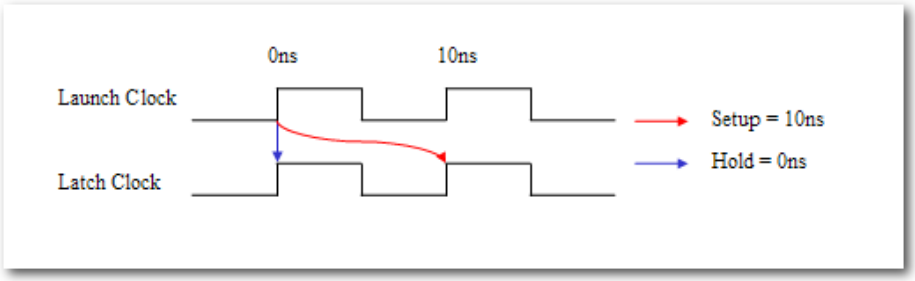


As can be seen, the Data Arrival Path starts with the Launch edge at 0ns and adds all the delays until it gets to the destination register. This results in a total delay of 3.875. The Data Required Path starts at time 10ns and goes through the latch clock's delay to the destination register, ending in 13.045ns. This meets timing since the Data Arrival Path is less than the Data Required Path, and it's 8.963ns less, which is the slack on this path.

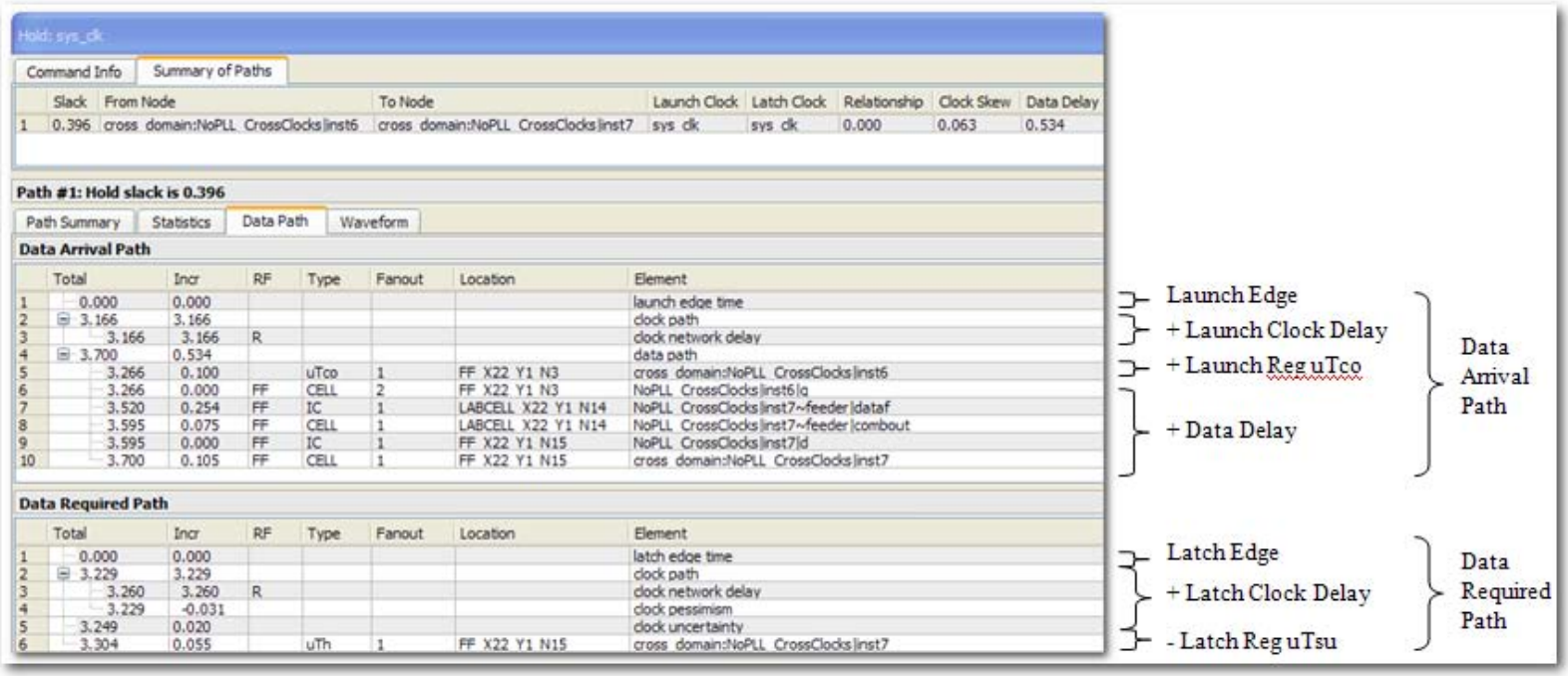
(Note that the clock path's are a single line item. This is because I ran *report_timing* with the option *-detail path_only*. If it were *-detail full_path*, then the clock tree would have been broken out in more detail. This is explained in the report_timing section of Getting Started.)

The Waveform tab also shows this information, although at a higher-level. I find the two tabs work together nicely, where the Waveform view helps users understand what is going on, and if the path fails timing, the Data Path tab helps detail why it fails, giving individual delays, placement information, Interconnect Delalys(IC), and Cell Delays(Cell).

Let's now look at the hold relationship of the same path. Going back to the waveform:



As can be seen, the hold relationship is 0ns, i.e. the latch edge is now at 0ns, and we want our data to arrive after the latch edge in order to meet timing. Looking at the same path above in a timing report:



Note that the differences between the setup and hold analyses are:

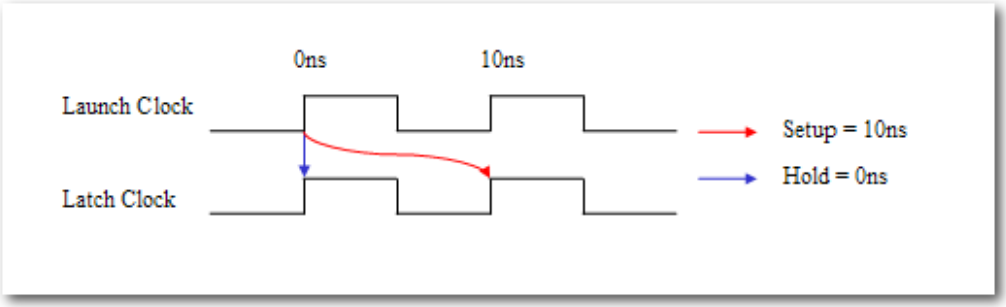
- The Launch and Latch Edges change. In this case the Launch Edge stayed the same, but the Latch edge went from 10ns to 0ns
- µTh is the micro parameter used for the Latch register instead of the µTsu.
- The Data Arrival Path is supposed to be greater than the Data Required Path. The slack is Data Arrival Path - Data Required Path, while for setup it was Data Required Path - Data Arrival Path. (This one is not apparent to everyone, but makes sense when you go back to the original waveform.)
- The delays vary slightly. This is due to rise/fall variation, on-die variation, and other timing model effects. For setup checks we wanted to compare the longest possible Data Arrival Path to the shortest possible Data Arrival Path, while for hold checks we want the shortest possible Data Arrival Path compared to the longest possible Data Required Path.

Hopefully these equations make sense and the user feels comfortable looking at their own paths and analyzing the results. Note that most paths are internal to the FPGA, have the same source and destination clock(are in the same clock domain), and that clock is on a global. When these conditions exist, the launch and latch clock delays are close to equal and subtract out of the equation, leaving the data path delay as the major component. This is what users often think of for static timing analysis, whereby if they have 10ns clocks, the data delay must be greater than 0ns and less than 10ns. This is a simplistic approach, but approximately correct when the clock delays are balanced.

Now that we've looked at how the initial clock waveform's default setup and hold relationship are used to analyze a path, let's find out how those default setup and hold relationships are calculated.

Default Relationships

By default, all clocks are related in TimeQuest and hence have a default setup relationship and hold relationship. This is easy to see when the clocks are straightforward, such as the following which have the same period and are edge-aligned:



Most paths fall into this simple relationship, but it is important to understand how default relationships are calculated for anything more complicated. Examples would include clocks with different periods, clocks with phase-shifts or offsets, registers clocked on the falling edge, etc.

Determining Default Setup and Hold Relationships in Three Steps

There are three simple steps for determining default setup and hold relationships:

1) Draw clock waveforms based on SDC constraints

2) The default setup relationship comes from the closest edge pairs where Launch Edge < Latch Edge

3) The default hold relationship comes from the closest edges where Launch Edge + Setup Relationship < Latch Edge

4) Optional - Verify/Validate in TimeQuest

Note that I use equations for steps 2) and 3), but rely on the waveforms to really determine the relationships, as we'll see. Let's go through these steps in more detail:

1) Draw clock waveforms based on SDC constraints

This is the step most users want to skip. Waveforms seem simple and the user can picture them in their head, but note that I use TimeQuest every day, and still find benefit in drawing out waveforms, no matter how simple they may be. So taking some SDC constraints:

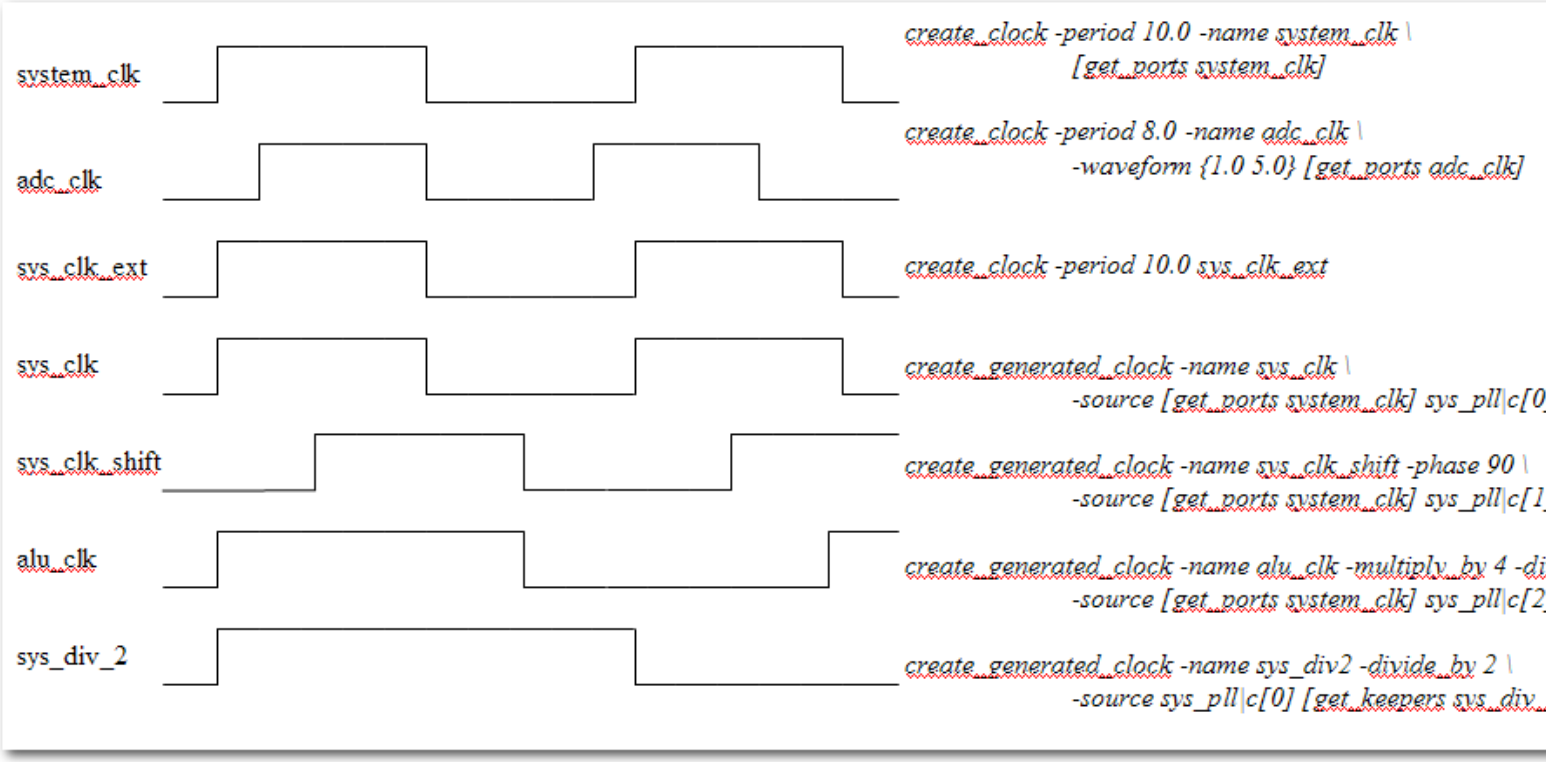
```
create_clock -period 10.0 -name system_clk [get_ports system_clk]
create_clock -period 8.0 -name adc_clk -waveform {1.0 5.0} [get_ports adc_clk]
create_clock -period 10.0 sys_clk_ext
derive_pll_clocks
        Info: Calling derive_pll_clocks {
        create_generated_clock -name sys_clk \
                -source [get_ports system_clk] sys_pll|c[0]
        create_generated_clock -name sys_clk_shift -phase 90 \
                -source [get_ports system_clk] sys_pll|c[1]

        create_generated_clock -name alu_clk -multiply_by 4 -divide_by 5 \
```

```
        -source [get_ports system_clk] sys_pll|c[2]
    }
create_generated_clock -name sys_div2 -divide_by 2 \
    -source sys_pll|c[0] [get_keepers sys_div_reg]
```

I show *derive_pll_clocks* since I recommend having that in the .sdc, but then show what generated clocks were created from that, copied from the TimeQuest messages. I also shortened the PLL names for readability. The final generated clock is on a divide-by-2 register in the design.

Anyway, drawing out the waveforms shows:



The purpose of this is not to show how to draw waveforms. Instead, let's take note of a few things:

- The waveforms are not dependent on the whether they are from *create_clock* or *create_generated_clock*. System_clock comes in on a port, while sys_clk is the output of a PLL, but their waveforms look the same.

- The waveforms are not dependent on their target. Clock sys_clk_ext is a virtual clock that is not applied to any target, yet has the same waveform as system_clk and sys_clk. Likewise, sys_div_2 is applied to a ripple clock register in the design, yet it's waveform is aligned with the other clocks.

- Only explicit options in the .sdc affect the waveform. Clock adc_clk has a -waveform option that offsets it by 1ns. Clock sys_clk_shift has -phase option that shifts it 90 degrees. Clocks alu_clk and sys_div_2 use -multiply_by and -divide_by that affect the waveform. Nothing from the user's HDL affects what the clock waveform looks like. Obviously the clocks

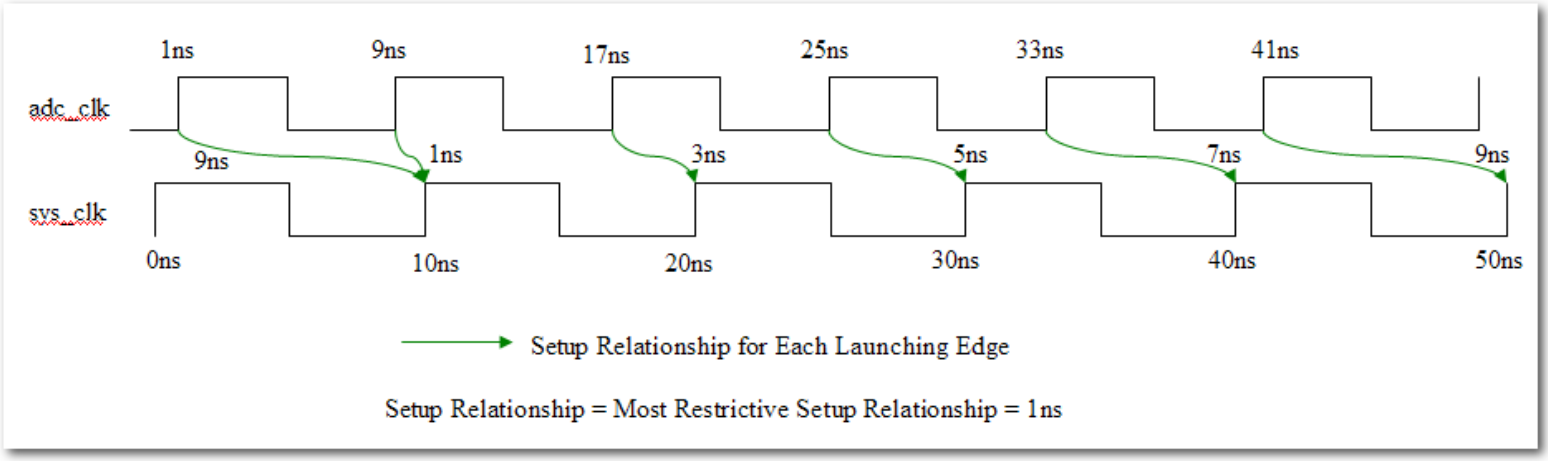
in the .sdc should match the clocks in hardware, the point is that making changes in the hardware will not change these waveforms.

Now that we've drawn the waveforms, let's go to step 2:

2) The default setup relationship comes from the closest edge pairs where Launch Edge < Latch Edge

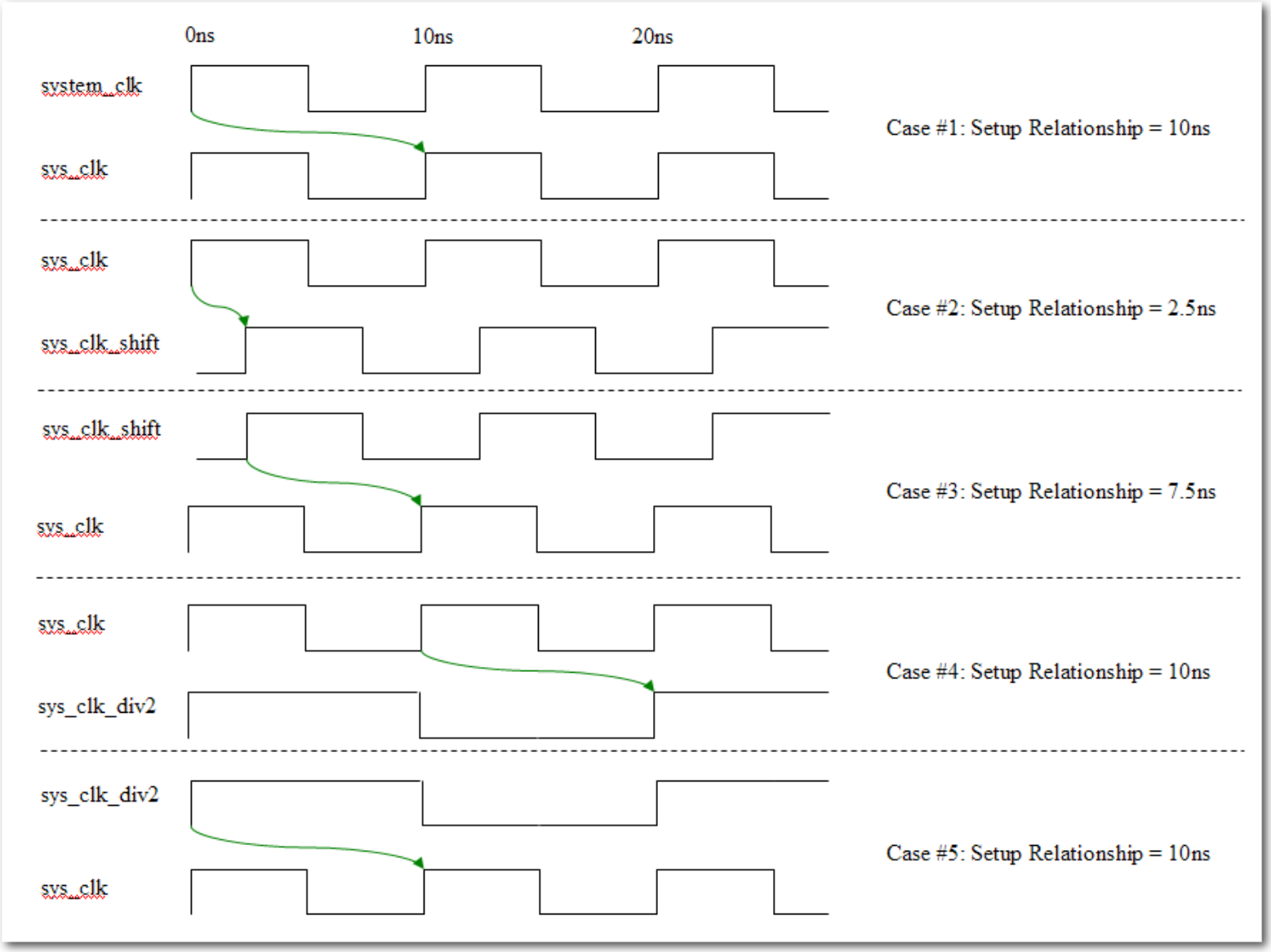
This is an equation, but easier to do from our waveforms. In essence, assume every edge launches data. Start with the first launch edge in your waveform and move forward to the nearest latch edge AFTER the launch edge. A difference as littls as 1ps counts. Then go to the next launch edge and repeat. Continue until a pattern shows up(values start repeating). The smallest latch-launch value is the default setup relationship.

Let's look at a complicated example, where a register clocked by adc_clk feeds another register clocked by sys_clk. As shown above, adc_clk is an 8ns clock with a 1ns offset, and sys_clk is a 10ns clock. Finding the default setup relationship looks like so:



The waveforms were drawn per step 1, then a line was drawn from every launch edge to the nearest latch edge after it. The default setup relationship is the smallest of these lines. So in this example, any transfers from adc_clk to sys_clk will default to a 1ns setup relationship. Note that we're not saying the other relationships don't matter, but that if we can meet the 1ns relationship then we've automatically met the other setup relationships. Of course, 1ns might be too tight of a requirement, and we will shortly be analyzing exceptions, which tell TimeQuest that the default relationship is not correct.

Now, the two clocks above are definitely strange, and most designs wouldn't transfer data between them. Most transfers are between clocks that are edge-aligned, or perhaps have a manual phase shift. Some common examples:



Case #1 is just a 10ns clock edge-aligned with another 10ns clock. In the original constraints above, system_clk is what comes in the FPGA input port while sys_clk is this signal going through the PLL. Even though the clocks are created differently and applied to different targets in the design, they still have a 10ns setup relationship. Only once the placed and routed design is analyzed will the skew between these clocks be analyzed.

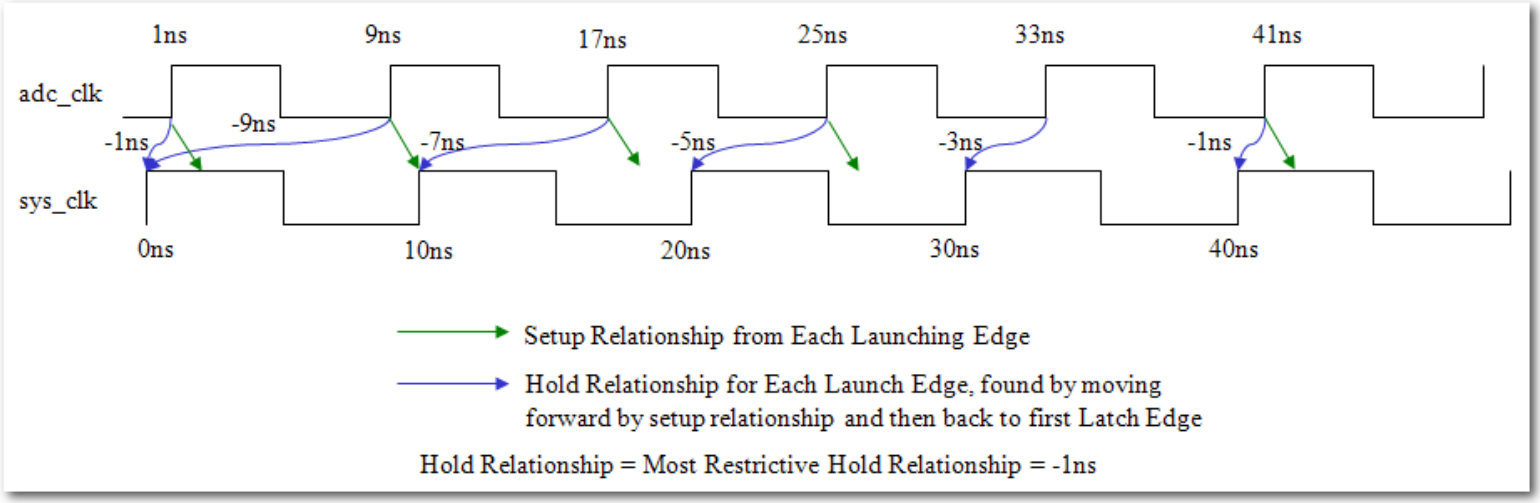
Cases #2 and #3 deal with sys_clk and sys_clk_shift, which is the same period as sys_clk but phase-shifted 90 degrees, or 2.5ns. When the latch clock is phase-shifted forward, the amount of that phase-shift amount of 2.5ns becomes the setup relationship. When the launch clock is phase-shifted forward, then (period - phase-shift) becomes the setup relationship.

Cases #4 and #5 deal with transfers between edge-aligned clocks, where one clock is a multiple of the other clock. In each case, the period of the faster clock becomes the default setup relationship.

These are just common examples. If ever unsure, follow Step 2) for determining the default setup relationship.

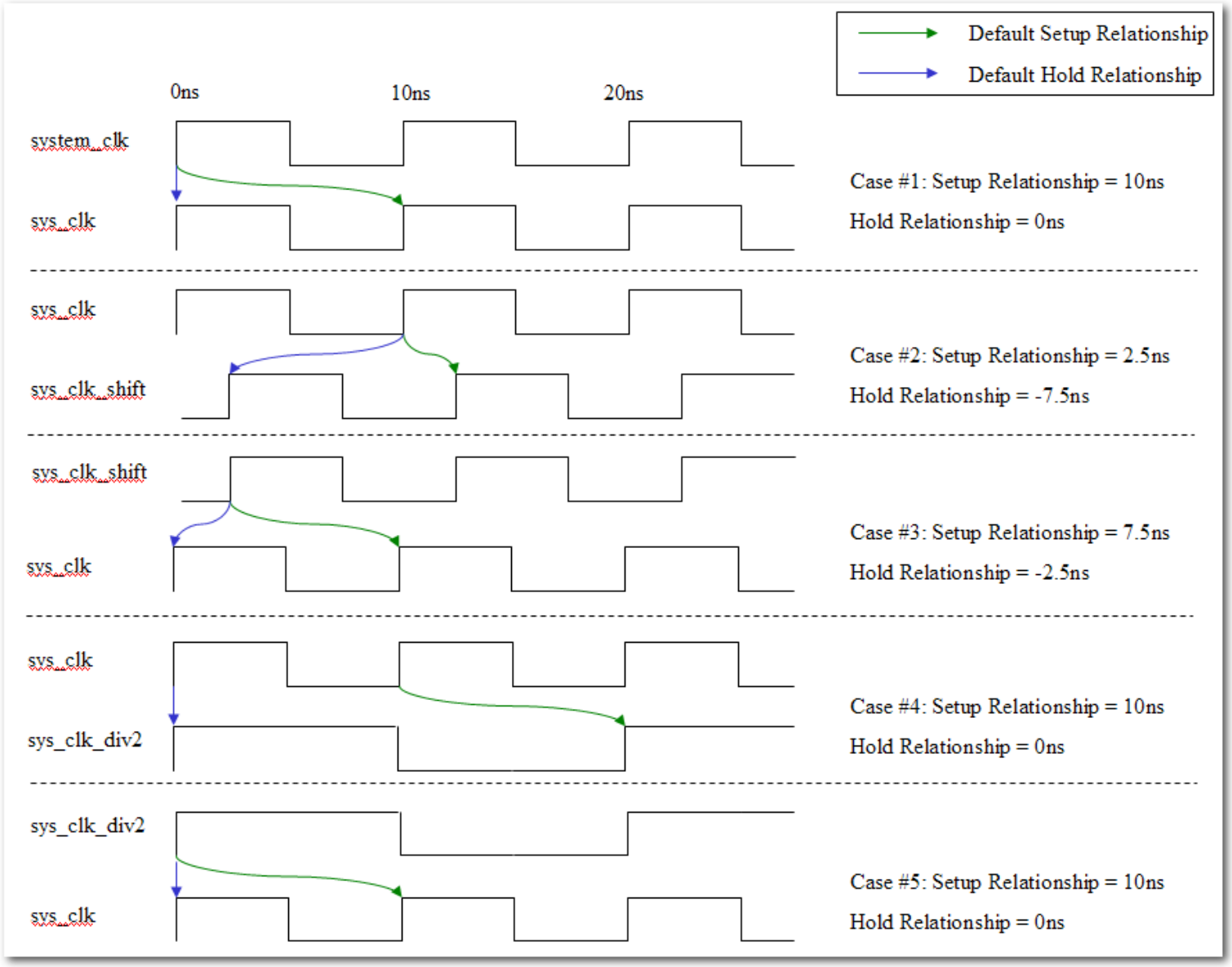
3) The default hold relationship comes from the closest edges where
Launch Edge + Setup Relationship < Latch Edge

Again this is an equation, but let's look at it from the waveforms. We have drawn them out and determined the most restrictive setup relationship. For the hold relationship, we will similarly assume that every launch edge sends data. So start with the first launch edge in the waveform, moving forward by the amount of the Setup Relationship, and then look for the first Latch Edge before that. Let's take our previous clock transfer used for the setup relationship:



In the diagram we start at each launch edge and move forward by the setup relationship(straight green arrow). We then move back to find the first latch edge before that(curving blue arrow). The difference between the launch edge and latch edge is the hold relationship, and TimeQuest will use the most restrictive one. Note that for hold, the most restrictive is the largest number, since we need to make sure the data arrival path is larger than the data required path by the hold relationship.

Rather than strange clock relationships like this, most designs have related clocks like the following examples:



- Note that Cases #1, #4 and #5 all have hold relationships of 0ns. For transfers with aligned edges, the default hold relationship will be 0ns.

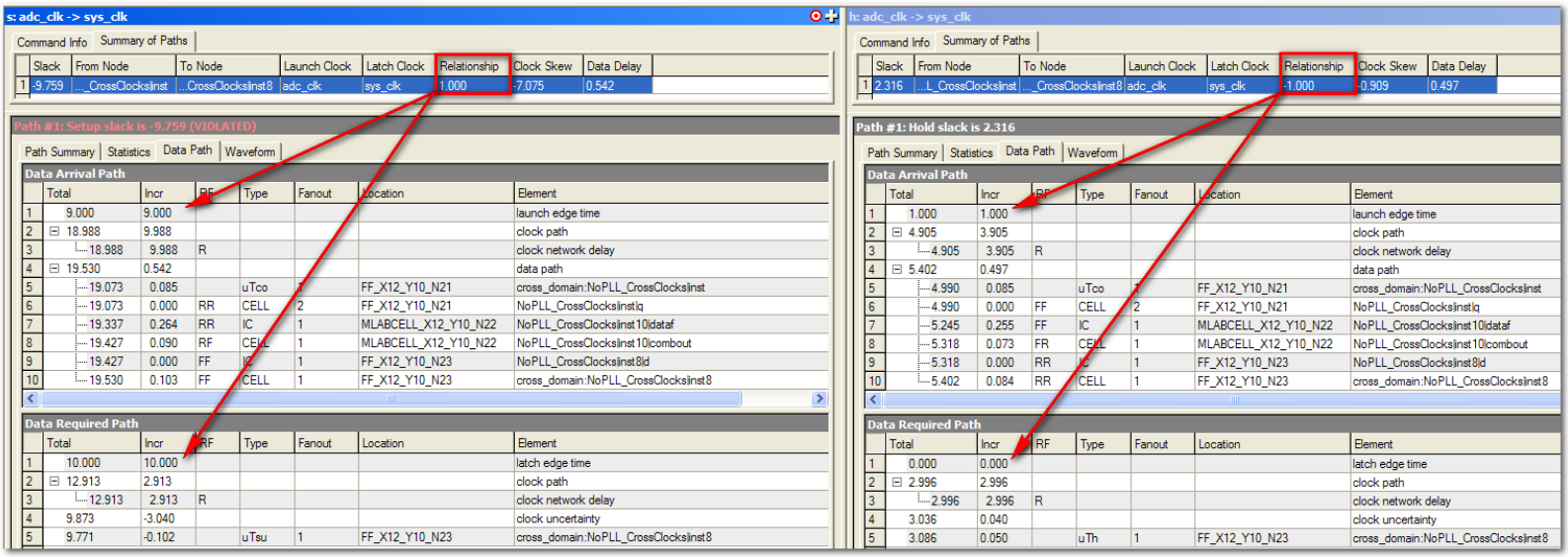
- The 0ns hold relationship is independent of the clock period. The clocks in Case #1 could have a period of 1 Hertz, and the hold relationship would still be 0ns. When designs have a timing failure, users often try to slow the clock down until it works. This is often valid for setup failures, but hold failures are generally immune and will fail at any frequency.

- Case #4 has different launch edges used for the setup relationship and the hold relationship. There is no requirement that the launch edge be the same edge for setup and hold analysis, it just works out that way most of the time. Remember that we assume all edges launch data and all edges latch data. If the user only looked at the launch edge at 10ns, they would determine the most restrictive latch edge to be at 0ns, for a hold relationship of -10ns. By looking at the other launch edges, specifically 0ns, we find a more restrictive hold relationship of 0ns, that still meets our requirement of being less than the setup relationship.

- Cases #2 and #3 are phase-shifted clocks. Note that the setup relationship - hold relationship adds up to the clock period. This is not guaranteed, but for most clock relationships this is true. It also makes sense, as that would be the fastest rate at which data can be passed between these clocks.

4) Optional - Verify/Validate in TimeQuest

The previous three steps show how to determine the default setup and hold relationship. They are mainly for understanding, since TimeQuest will be doing this on its own and reporting it to the user. Since TimeQuest is doing this, all the user needs to do is run *report_timing* on a path between the specified clock domains to get the relationships. For the difficult case above, where the launch clock has an 8ns period with a 1ns offset, and the latch clock has a 10ns period, we calculated the default setup relationship to be 1ns and the default hold relationship to be -1ns. Running *report_timing -setup* and *report_timing -hold* between these clocks in TimeQuest shows:



The left timing report shows the setup analysis, where the setup relationship is 1ns. The launch edge time is 9ns and the latch edge time is 10ns. Likewise on the right panel we see the hold analysis, where the hold relationship is -1ns, shown with a launch edge time of 1ns and latch edge time of 0ns.

Now, we already knew this would be the setup and hold relationships based on our analysis, but it's good to see the correlation with TimeQuest. Most importantly, if you're not completely sure how to calculate a relationship, you can always have TimeQuest do it for you.

Points of Interest for Default Relationships

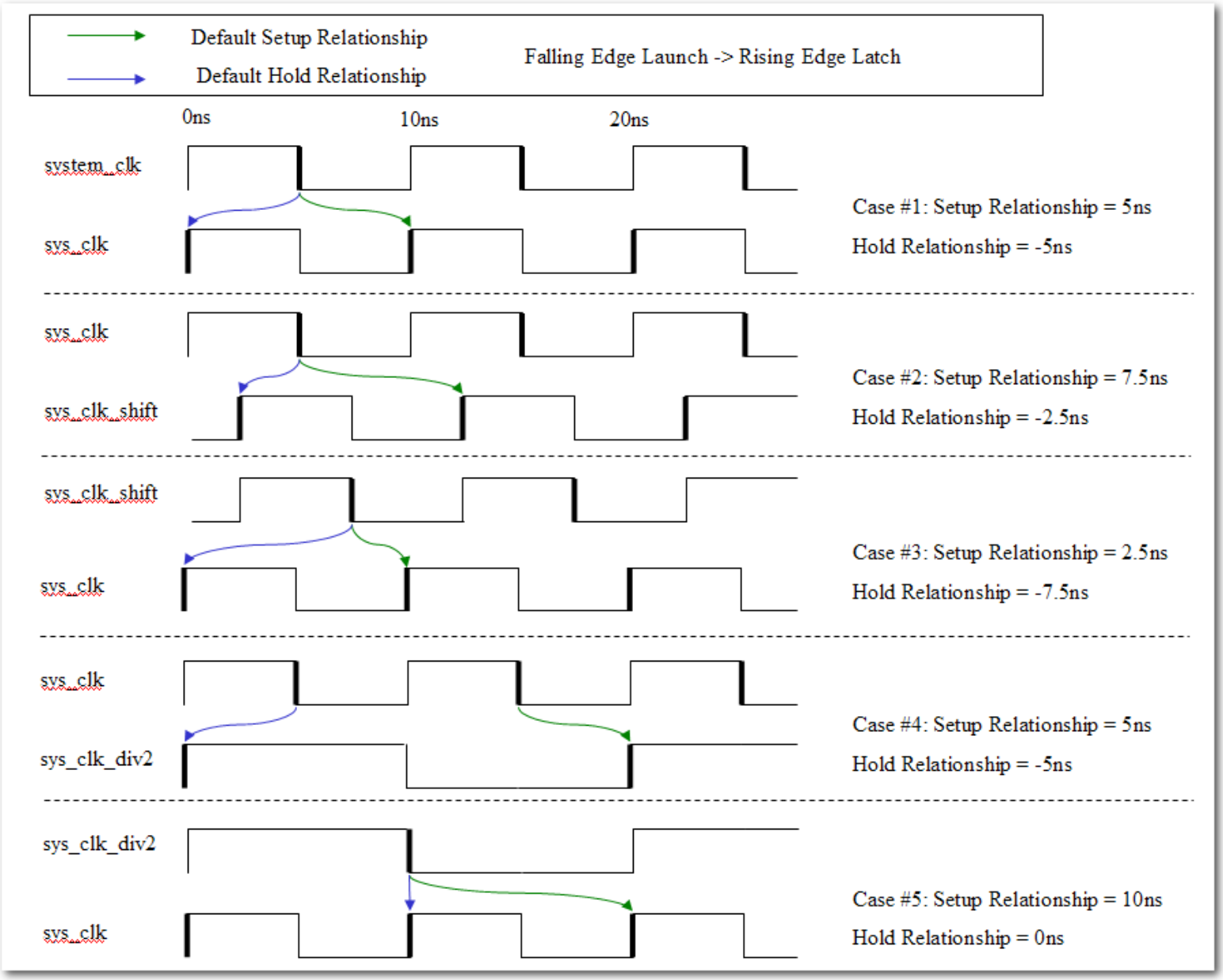
Now that we know how to determine default setup and hold relationships, there are some points of interest worth noting:

Falling Edge Analysis

Up until now, this section has assumed the registers are clocked on the rising edge, but the analysis can also be done for registers clocked on the falling edge. Note that the steps for determining setup and hold relationships should be run independently for these transfers. In fact, when a user defines two clocks, TimeQuest determine 16 different relationships between those clocks. For example, if we define sys_clk and adc_clk, TimeQuest determines:

sys_clk rising -> adc_clk rising setup relationship	sys_clk rising -> adc_clk rising hold relationship
sys_clk rising -> adc_clk falling setup relationship	sys_clk rising -> adc_clk falling hold relationship
sys_clk falling -> adc_clk rising setup relationship	sys_clk falling -> adc_clk rising hold relationship
sys_clk falling -> adc_clk falling setup relationship	sys_clk falling -> adc_clk falling hold relationship
adc_clk rising -> sys_clk rising setup relationship	adc_clk rising -> sys_clk rising hold relationship
adc_clk rising -> sys_clk falling setup relationship	adc_clk rising -> sys_clk falling hold relationship
adc_clk falling -> sys_clk rising setup relationship	adc_clk falling -> sys_clk rising hold relationship
adc_clk falling -> sys_clk falling setup relationship	adc_clk falling -> sys_clk falling hold relationship

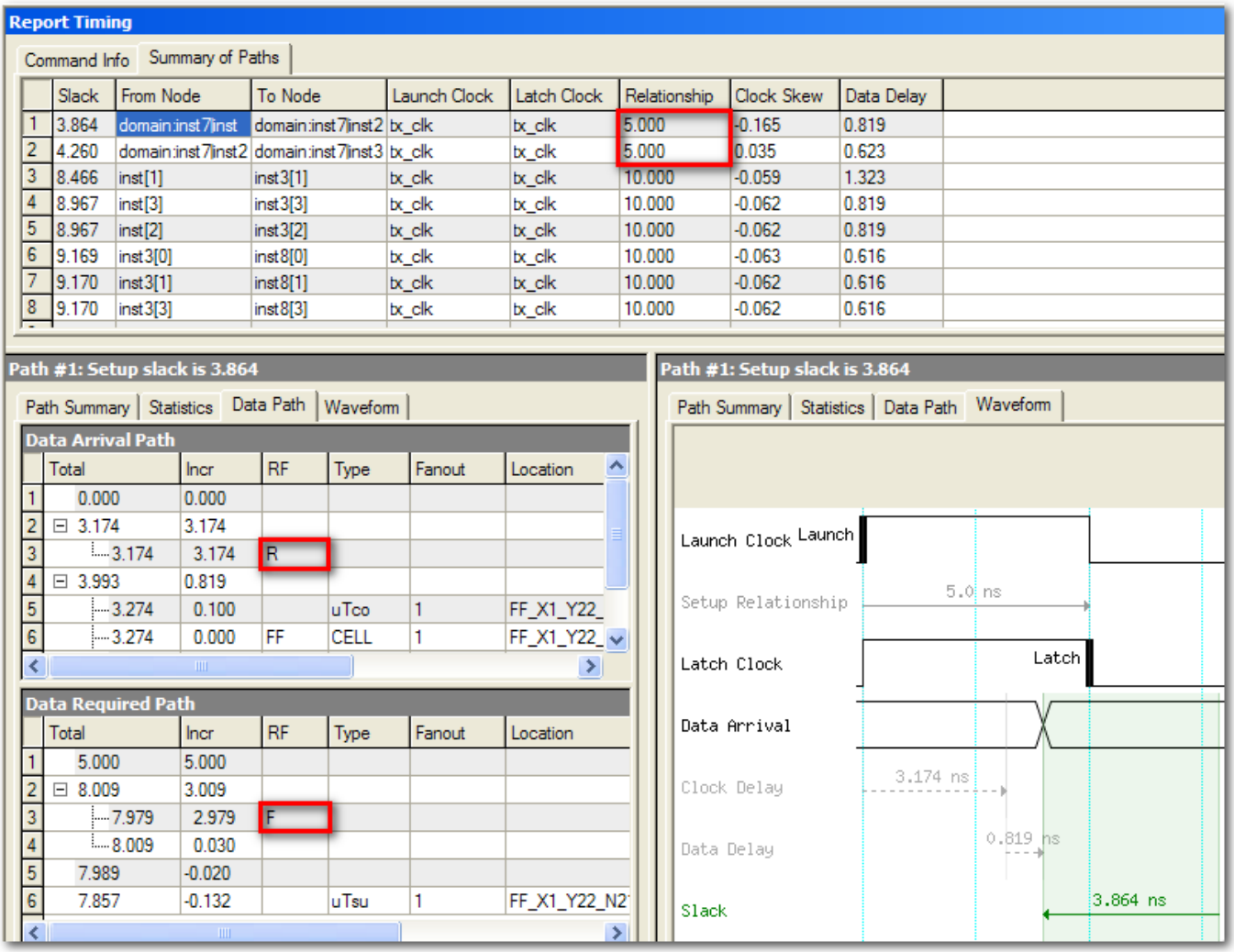
Now, in most designs these relationships will be the same for multiple scenarios. In determining the default setup and hold relationship for falling edge registers, just follow the same steps used, but the launch and/or latch edges should use the falling edge, depending on the situation. For example, let's look at the clock transfers we've been using, but re-analyze them when the source register is clocked on the falling edge:



Since we're analyzing a falling launch edge to a rising latch edge, the edge of concern have been highlighted. Most falling to rising edge transfers(or vice-versa) occur within a domain, and so most relationships are like Case #1, where the setup relationship is a half period and the hold relationship is a negative half period.

Also note that the Setup Relationship - Hold Relationship still adds up to the period of the faster clock. I also think, if I had this hooked up in my design, this is the relationship I would expect. When there are problems with falling edge registers, it's usually not that the user doesn't understand the default relationships, or that the defaults are not the user's intent, it's usually that they don't realize a register is clocked on the falling edge. This results in case #1 above, where the setup relationship is half the clock period, and they end up not meeting timing. There are

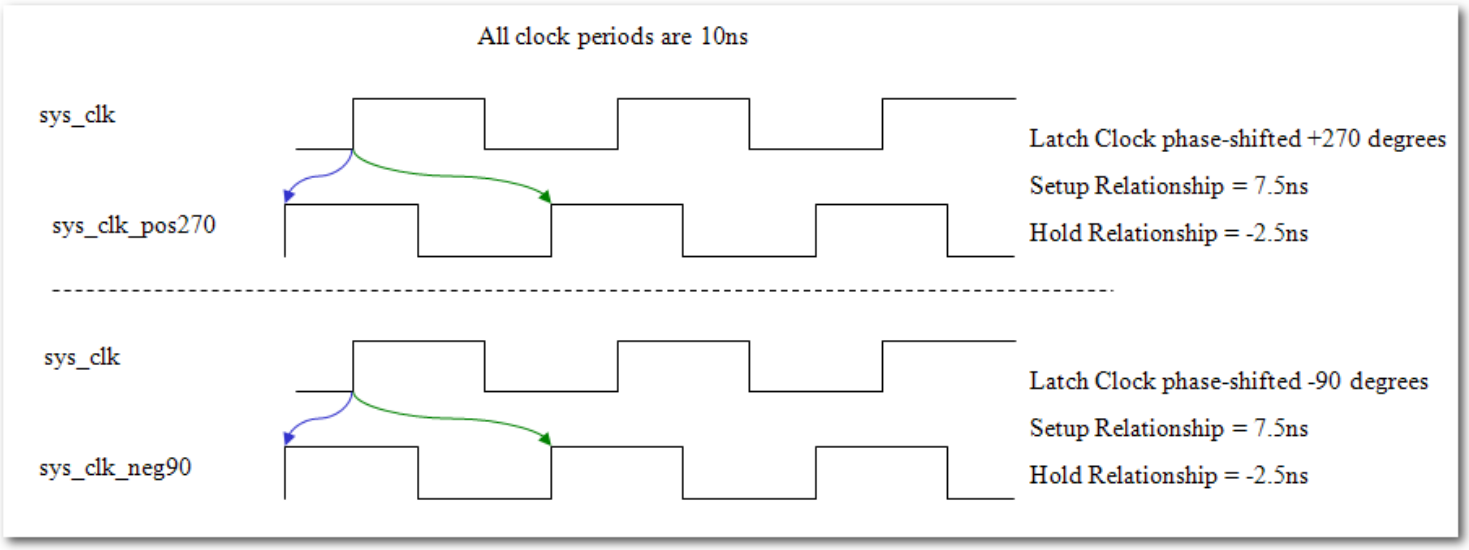
some identifiable points in TimeQuest. The following report timing is on a design where there are a chain of registers, and one in the middle is clocked on the falling edge:



As can be seen, two paths have a 5ns setup relationhip. This is the Rise->Fall transfer to the register, and the Fall -> Rise transfer from the register. In the Data Path tab, the launch clock is shown with R for a rising edge while the latch clock is shown with an F for falling edge. The Waveform tab also clearly identifies the Launch edge is rising while the Latch edge is falling.

Periodicity

The way to determine default setup and hold relationships requires the user to look at edges over time. Admittedly, in most cases the first edge or two is the correct one to use, but as we'll see in a moment with unrelated clocks, it may be many cycles out in time before the most restrictive setup or hold is found. The nice thing about this is that our waveforms are considered periodic, not just a single snapshot. For example, a designer could take a clock coming out of a PLL and phase-shift it +270 degrees or -90 degrees, and they would get the same relationships to other clocks:



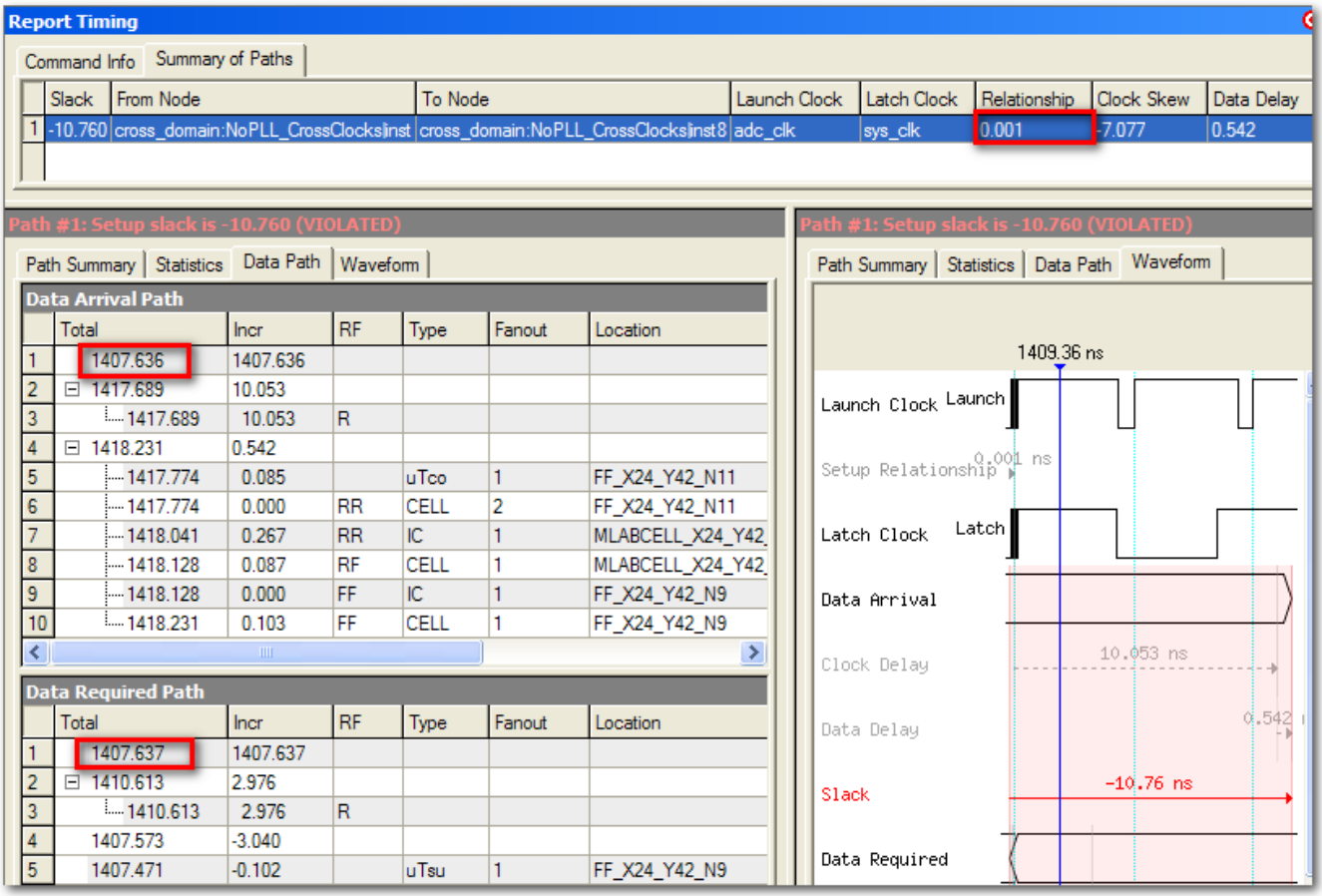
Other examples of periodicity are:

- Moving the launch clock back 90 degrees or the latch clock forward 90 degrees will result in the same relationships between those two clocks.
- Inverting a clock to a register or phase-shifting it +/-180 degrees results in the same relationships to other clocks.

This periodicity matches what occurs in hardware, so it's good to see timing analysis reflect that.

Relationships between Unrelated Clocks

What happens when two clocks are clearly unrelated? For example, what is the setup relationship if the launch clock has a 4.567ns period and the latch clock has a 7.777ns period? TimeQuest will do exactly what it is supposed to, and find the most restrictive setup relationship over time:



Following the procedure, it finds that after launching data at time 1407.636ns, there is a latch edge exactly 1ps after that, which becomes the setup relationship. The path naturally fails timing and shows up at the top of their list.

How is a user supposed to calculate that? They're not. In reality, these clocks can't be related and the user shouldn't care how TimeQuest relates them. Instead, the user should either be fixing the data path or applying a *set_clock_groups* or *set_false_path* assignment on the path or between the clocks, to tell TimeQuest not to analyze this path in a synchronous manner. The important point is being able to recognize why this occurs.

Some users rely on this phenomenon to find code problems. For example, if they mistakenly modify their RTL so there is a path from adc_clk to sys_clk, it will get a 1ps setup relationship, fail timing, and show up at the top of their list as a failure for that domain. They then analyze the path and either realize they need to synchronize properly between the domains, or apply a *set_false_path* directly on the path. The problem with this is that there is no guarantee the default setup relationship will be 1ps. For example, let's say the user has two independent 20ns clocks coming into the FPGA, which they constrain like so:

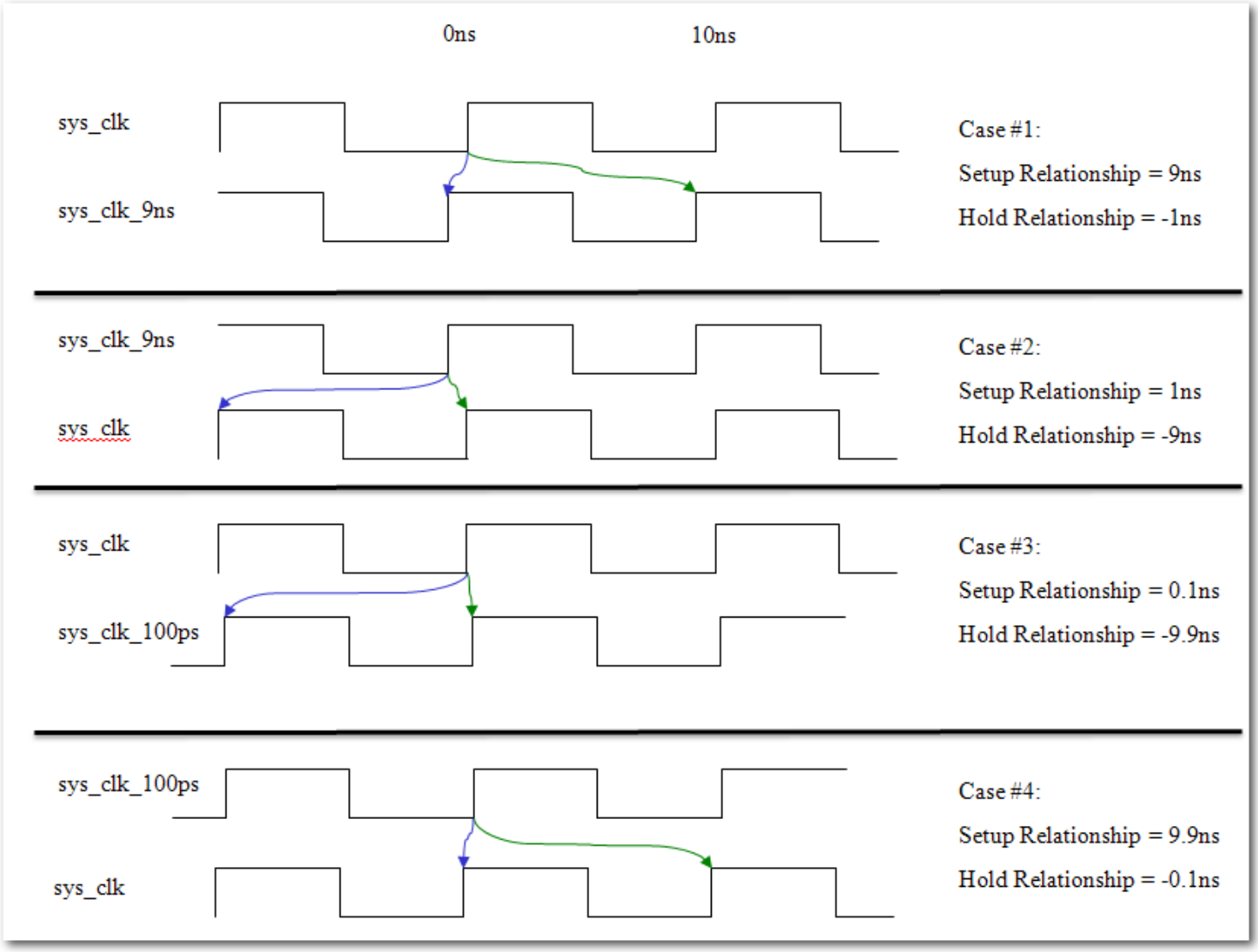
create_clock -period 20.0 -name clk_a [get_ports clk_a]
create_clock -period 20.0 -name clk_b [get_ports clk_b]

Now, if the clocks are from independent sources, the will have no known phase-relationship and will vary from each other by some parts-per-million(PPM) difference. In essence, it will be impossible to synchronously pass data between these clock domains. But if a

user mistakenly passes a data bus from clk_a to clk_b, TimeQuest will see a default setup relationship of 20ns, the fitter will try to meet timing on those paths, and assuming it can, the paths will not show up as a timing failure. So in this case, relying on a tight relationship between unrelated clocks to identify mistakes in the code did not work.

Phase-Shift Affect on Setup and Hold

Manually phase-shifting a clock, usually done with a PLL, naturally affects the setup and hold relationships to other clocks, but it is important to understand exactly how. Let's look at some quick examples usingthree 10ns clocks, one without any phase-shift, one with a 9ns phase-shift, and one with a 100ps phase-shift.



Case #1 shows transfers from the base clock to the 9ns phase-shifted clock. The setup relationship is 9ns and the hold is -1ns. This makes sense and probably what the user wants. But when we go to Case #2, which are transfers in the other direction, the setup relationship is 1ns

and the hold relationship is -9ns. This is probably not what the user wants. Note that there may not be any transfers in this direction, in which case they don't care what the relationships are, but if there are, a multicycle may be necessary.

As mentioned, clocks are periodic, so a 9ns phase-shift has identical relationships as a -1ns phase-shift.

Case #3 is even more extreme, whereby the latch clock is phase-shifted by a mere 100ps. Since the default setup relationship is the most restrictive latch edge after a launch edge, the setup relationship is now 100ps. If the clock was phase-shifted as little as 1ps, then that would be the setup relationship. Again, this is probably not what the user wants, and a multicycle may be necessary.

Important Note: When the user phase-shifts their clock, they should determine the relationship to other clocks and determine if they need a multicycle to shift the window data is passed through. Multicycles are discussed in the next section, including this specific scenario where the user wants to shift the window

Of course, a phase-shift does not always mean a multicycle is necessary. Let's say the clock was phase-shifted 180 degrees. The default setup and hold relationship to the unshifted clock would be 5ns and -5ns, which is probably what the user wants.

New users often think that, if they phase-shift a clock 100ps, TimeQuest should be able to figure out that they don't want that to be the setup relationship and should target the next edge. By itself, this is probably true. The question does come up of determining what phase-shift is obviously not targeting the next edge? 90 degrees? 180 degrees? More importantly, there are a number of technical reasons that make TimeQuests's methodology correct. It preserves periodicity. It allows generated clocks of generated clocks, each of which have phase-shifts. It allows various clocks from various sources and their generated outputs to all have easily determined relationships. In the simple case of a single phase-shifted clock with a small shift, it may look silly, but for a robust timing engine, it is correct.

Multicycles

Now that we've examined all the ins and outs of default setup and hold relationships, it's time to examine multicycles, which are the main way users tell TimeQuest to use a relationship other than the default. Multicycles are based on existing edges defined by the clocks, and just tell TimeQuest to use a non-default launch or latch edge.

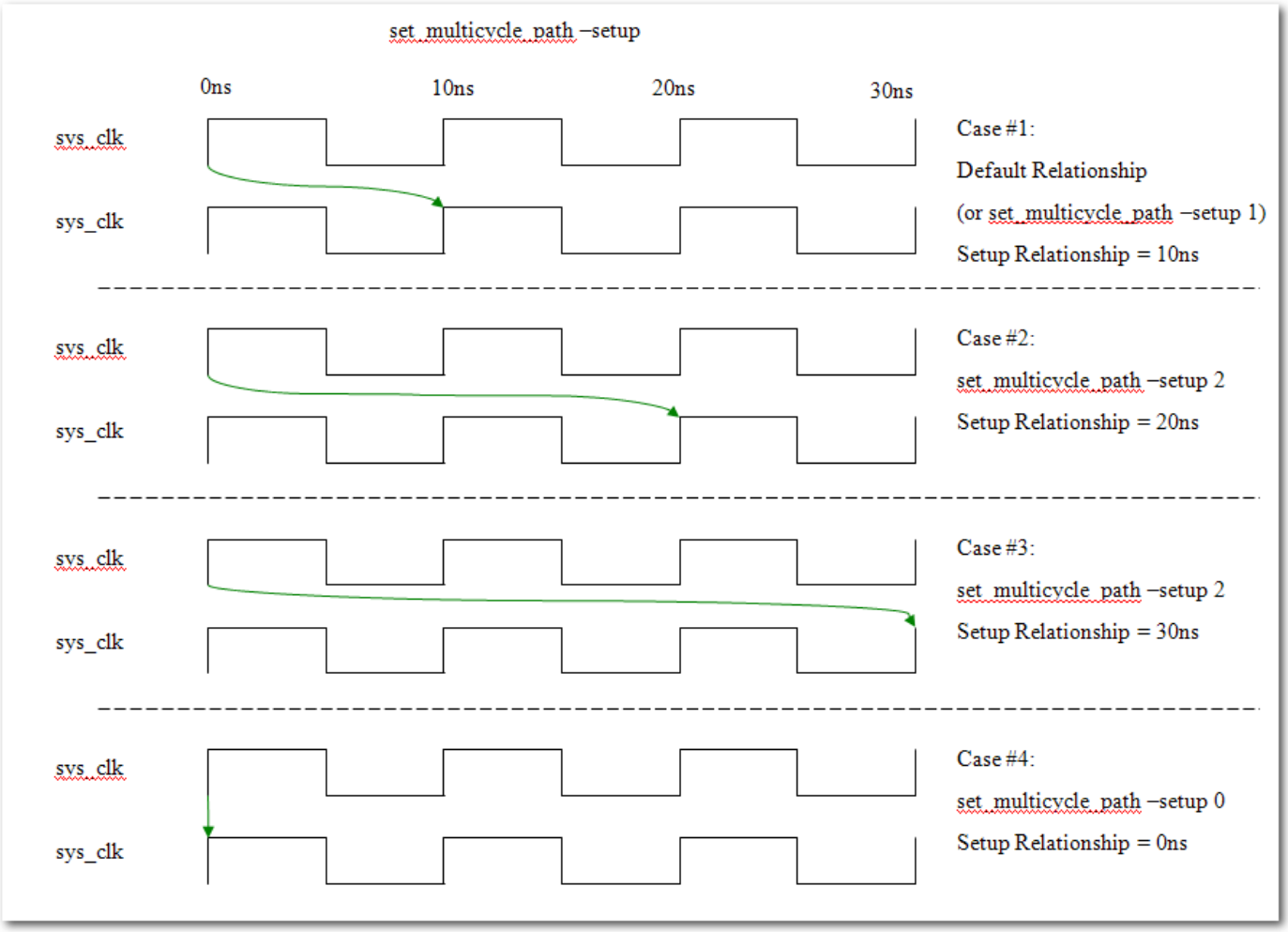
A major benefit of using multicycles is that they work in conjunction with the clock assignments. Since multicycles are based on existing clock edges, a user can modify their clock constraints and all the multicycles should adapt accordingly. This phenomenon is described here.

Note that most multicycles fall under two common cases, and if users know those, they can get by without understanding the ins and outs of multicycles. Those are explained at the end of this section, and may users could just skip to that part. Also, step 6) below just points out that TimeQuest will always tell you what a multicycle does to a relationship, so it's certainly possible to get by without understanding how multicycles are calculated, and just follow step 6) whereby the user guesses at a multicycle value, looks at what TimeQuest calculates it to be, and the user determines if that is what they want or if they should try a new multicycle value.

Determining Multicycle Relationships in Five Steps

1) Draw clock waveforms based on SDC constraints
2) The default setup relationship comes from the closest edge pairs where Launch Edge < Latch Edge
3) **Apply multicycle –setup modification**
4) The default hold relationship comes from the closest edges where Launch Edge + Setup Relationship < Latch Edge
5) **Apply multicycle –hold modification**
6) Optional - Verify/Validate in TimeQuest

Note that steps 3) and 5) are new. The other steps are identical to the steps in determining the default setup and hold relationships. We're going to skip steps 1) and 2) since they have been covered and go right to step 3). An important point is that the default setup relationship is considered the "1" edge. If a multicycle applies a larger number, then the setup relationship gets larger(easier to meet). Let's look at some examples:



Case #1 is the default setup relationship. If a user applied a multicycle -setup 1, they would get the same results, which is why the default is called the 1 edge. As the multicycle value gets larger, the setup relationship grows by that many clock periods. So in Case #2, with a

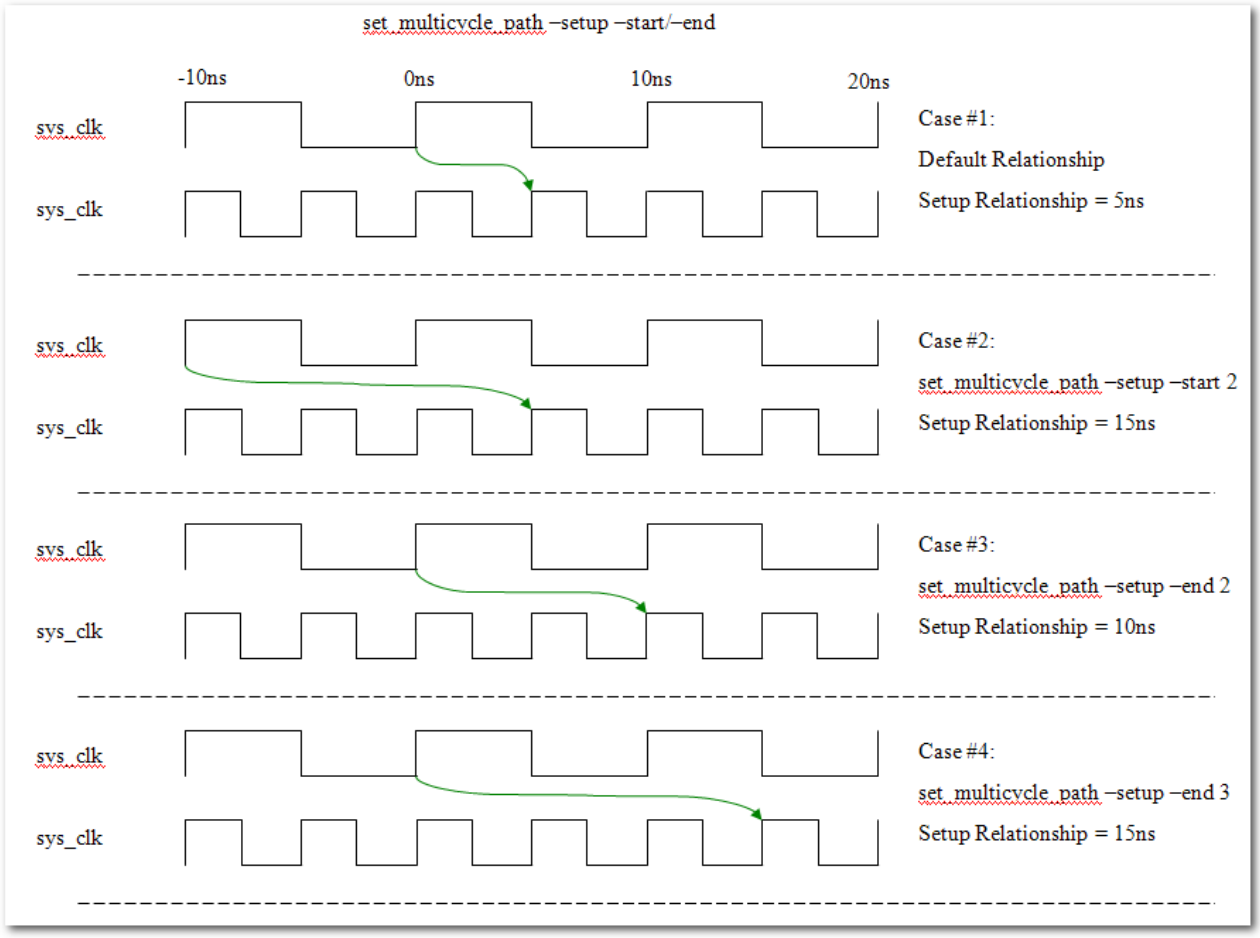
multicycle -setup of 2, the setup relationship is 20ns. Case #3 takes it to 30ns. I put in Case #4, where the multicycle setup is 0, and since the value is 0, the setup relationship is reduced by a clock period to 0ns. This is occasionally what a user wants in source-synchronous interfaces, which is why I point it out. In essence:

*setup relationship = default_setup_relationship + (MC_setup_value – 1) * clk_period*

So in Case #2, we start with the default relationship of 10ns. A multicycle of 2 was applied, so our new setup relationship is 10 + (2-1)*10 = 10 + 10 = 20ns.

As discussed in the section on *set_multicycle_path*, the actual constraint could be applied between nodes in the design, or between clocks. The -from/-through/-to options determine what paths the assignment is applied to, while the -setup <value> determines how much the assignment modifies the default relationship by.

Now, in the equation above we use the term *clk_period*. What if our launch and latch clocks have different periods? This is determined by the option -start/-end. If no option is given, *set_multicycle_path* defaults to -end. This option determines whose clock period to use, whereby -start means to modify the relationship by the period of the launch clock, and -end means to modify the relationship by the period of the latch clocks. Another way to think of this is to begin with the default setup relationship, and if the option is -start, move the start of the arrow back in time that many edges, and if the option is -end(the default if no option is specified), move the end of the arrow forward this many edges. Some examples:



Case #2 has a multicycle -setup -start 2. Since we use the -start, we're going to move the start of the green arrow back one cycle, so our setup relationship increases by the period of the launch clock, from 5ns to 15ns. Case #3 is a multicycle -setup of 2, but the -end option is specified(and the default if -start/-end is not specified), and the end of the arrow is moved out a clock cycle, taking the setup relationship from 5ns to 10ns. Case #4 takes it to 15ns. So in this particular example, where the destination clock is half the period of the source clock, a multicycle -start 2 and a multicycle -end 3 both result in a setup relationship of 15ns. Remember that our relationship is the difference between the launch and latch edges, so Case #2 and #4 have the same relationship, even though the arrow is drawn between different edges.

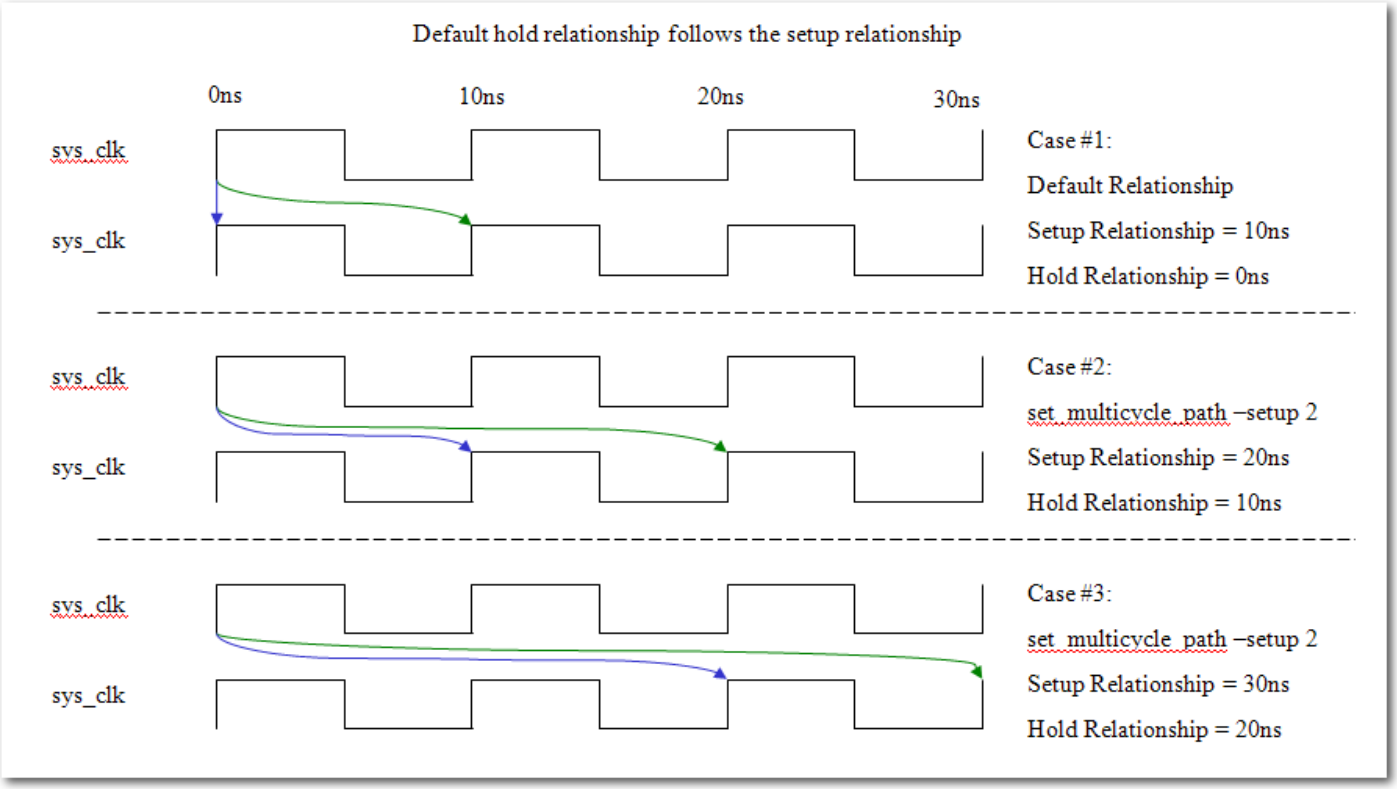
From an equation perspective, we need to introduce some logic, making the calculation look something like the following, whereby -start and -end are mutually exclusive options:

*-start setup relationship = default_setup_relationship + ((MC_setup_value – 1) * launch_clk_period)*
or
*-end setup relationship = default_setup_relationship + ((MC_setup_value – 1) * latch_clk_period)*

Now that we've done the newly added step 3), let's look at step 4):

4) The default hold relationship comes from the closest edges where Launch Edge + Setup Relationship < Latch Edge

We already did this step when determining default relationships, but note that step 4) comes **after** the multicycle -setup is applied, so the default hold relationship follows the setup multicycles, like so:



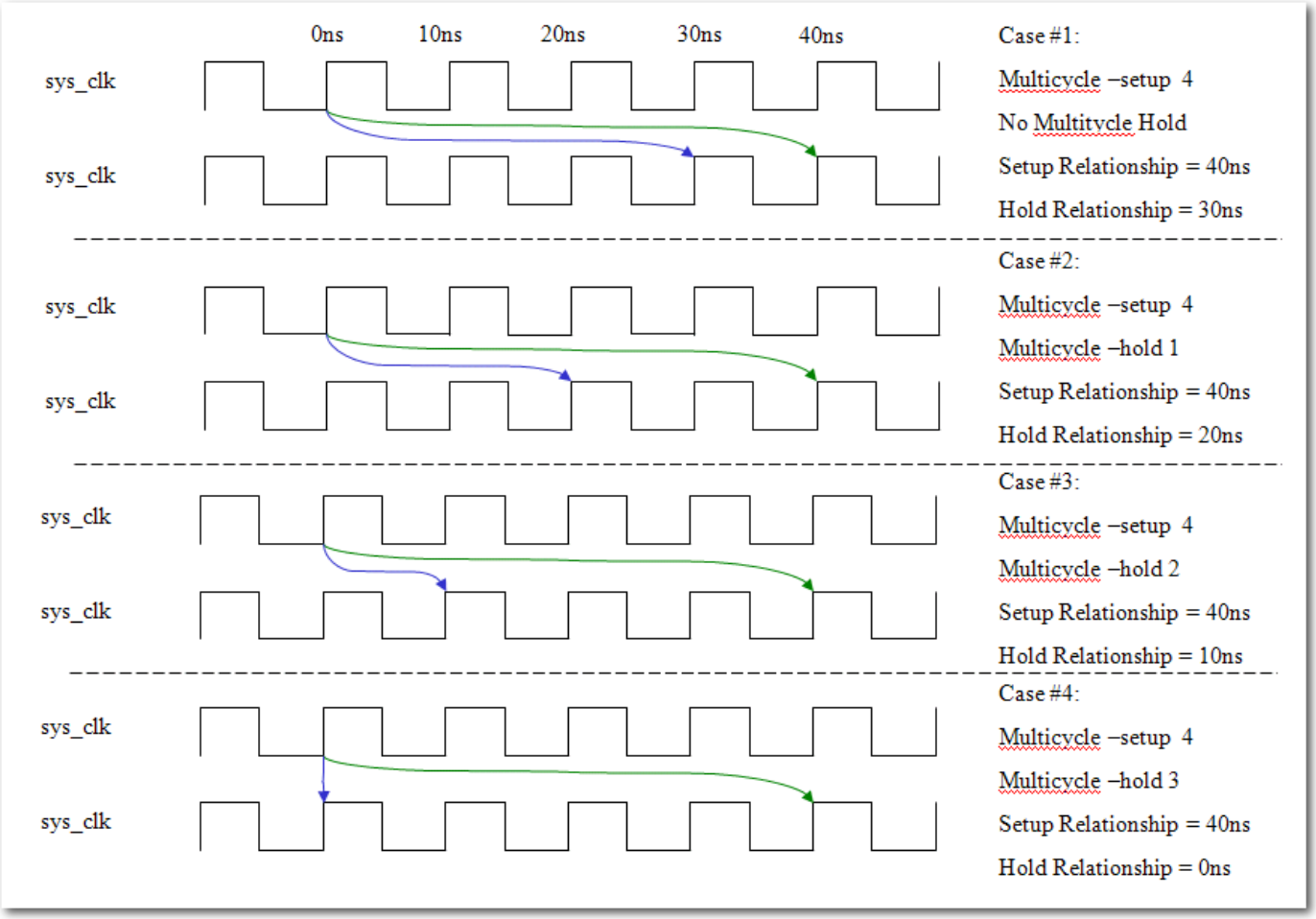
As can be seen in Case #2, after applying a multicycle -setup 2, not only does the setup relationship increase to 20n, the default hold relationship increases to 10ns. This is because the default hold relationship is determined from the setup relationship, even if the setup relationship was modified with a multicycle. So following our methodology of determining default hold relationships, in Case #2 we assume every launch edge launches data. We start at the first launch edge at 0ns, go out by the setup relationship which is now 20ns, and start moving back until we find the first latch edge, which occurs at 10ns. This is our default hold relationship now.

Case #3shows this again, as the setup relationship goes to 30ns due to multicycles, the default hold relationship also increases to 20ns. As can be seen, the multicycle hold tends to shift the window data passes through but doesn't change how big it is. (By window I mean the difference between the setup relationship and hold relationship, where data passes through.) If the default hold relationship is not what the user wants, they can apply a multicycle -hold.

(Note: The default hold relationship follows the setup relationship with multicycles. If a user applies a *set_max_delay* constraint to override the setup relationship, the default hold relationship does NOT follow that. It is still based on the setup relationship, either the default or with multicycles).

5) Apply multicycle –hold modification

In step 4, the default hold relationship is called the "0" edge. As seen in the last set of waveforms, this default hold relationship changes with multicycle setups, but is still considered the "0" edge for each case. Applying a multicycle hold greater than 0 will loosen(make smaller) the hold requirement by that many edges. Multicycle holds are usually applied to paths that have a multicycle setup applied first, so we will look at a case with a multicycle -setup of 4:



As can be seen in Case #1, the default "0" edge for the hold relationship is at 30ns. As we apply multicycle holds that are greater than 0, the end of the arrow moves back in time, making the hold requirement smaller, and hence easier to meet. With a multicycle setup of 4, we need to apply a multicycyle hold of 3 to get the hold relationship back to 0ns.

Just like setup multicycles, when the launch and latch clock have different periods, the -start/-end option determine which clock's period to move the hold relationship by. The -start option moves the start of the hold arrow forward by one clock cycle, while the -end option moves the end of the arrow back by one clock cycle. Looking at it as an algorithm where the multicycle can only have -start or -end:

*-start hold relationship = default_hold_relationship - (MC_hold_value * launch_clk_period)*
or
*-end hold relationship = default_hold_relationship - (MC_hold_value * latch_clk_period)*

6) Optional - Verify/Validate in TimeQuest

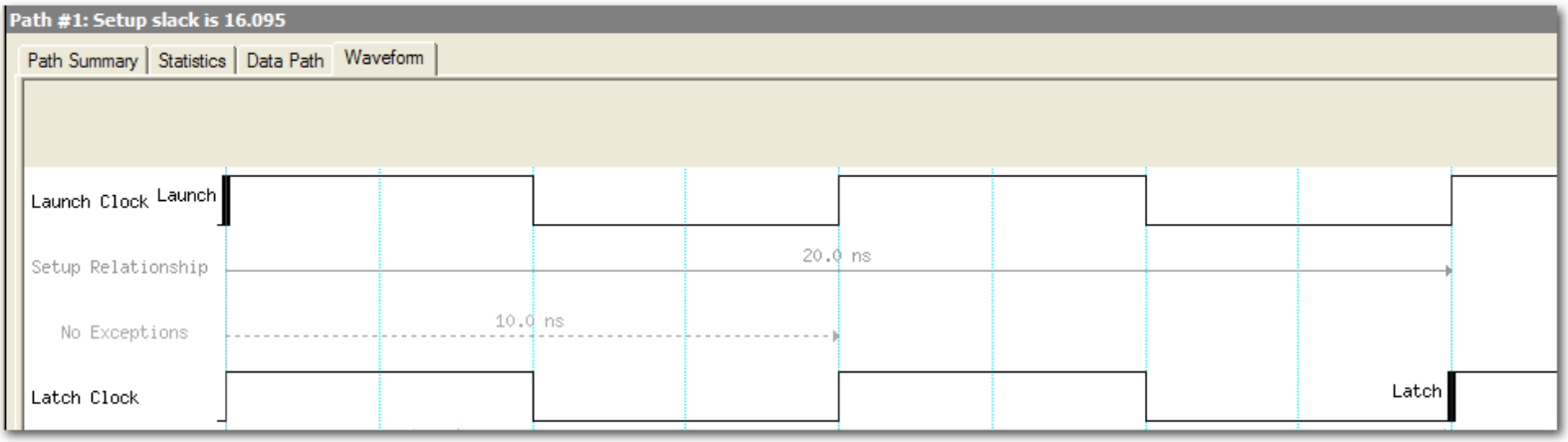
Just like with default relationships, TimeQuest's *report_timing* will explicitly report the setup and hold relationships it is using. So if you're unsure of what you're doing or just want to make sure your analysis is correct, quickly enter the multicycles you think are correct and do *report_timing -setup* and *report_timing -hold* on the path and see what setup relationship and

hold relationship TimeQuest calculates. If they are not what you want, modify the multicycle values and re-run TimeQuest.

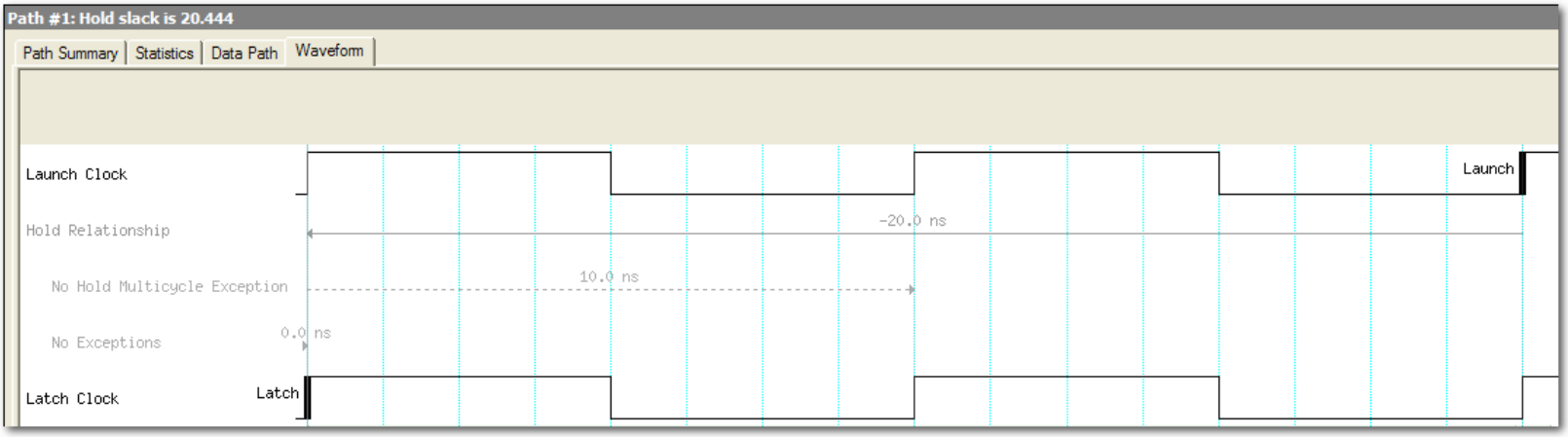
When analyzing a path, the waveform tab nicely shows the previous 5 steps. For example, let's say I have a 10ns clock and put in the following constraints:

*set_multicycle_path -setup -from {*source} -to {*dst} 2*
*set_multicycle_path -hold -from {*source} -to {*dst} 3*

For this example, let's not worry about why the user has these multicycles, but how they are reported. When I look at the *report_timing -setup* waveform tab:



Step 1 was to draw the waveform, which is done above. Step 2 is to determine the default setup relationship. That is the dotted arrow labeled No Exceptions, and shows what the setup relationship would have been without a multicycle setup. This is purely informational, as the Setup Relationship arrow shows the relationship after the multicycle setup of 2 is applied. This is what is used for the anlaysis. Now let's look at the *report_timing -hold* waveform tab:



The No Exceptions is what the hold would be if there were no setup or hold multicycles. The No Hold Multicycle Exceptions is step 4), where the hold relationship is determined based on the setup relationship, showing what it would be with just the multicycle setup but before the multicycle hold. Since the multicycle setup shifted the setup by one clock period, the hold

relationship has also increased by one clock period to 10ns. Both of these arrows are for informational purposes only.

Finally, we do step 5) and apply the multicycle hold of 3, which shifts the hold relationship by 3 clock cycles, moving it from +10ns to -20ns.

Note that if you had done all these steps with pencil and paper, you might end up with different edges then what the waveform viewer shows. For example, my final hold relationship started with a launch edge at 0ns and a latch at -20ns, while the one above launches at 20ns and latches at 0ns(I cut off the time scale). These different launch and latch times will give the exact same analysis and the exact same slack, since it's the difference between the launch and latch edges that we ever care about. Don't get hung up if TimeQuest chooses different edges as long as the difference between your launch and latch edges is the same.

Designing with Multicycles

I've found the previous steps for calculating multicycle relationships to be the best way to learn how they work, whereby the user can take different multicycle values and determine what the new relationships will be. The user may encounter this when given an .sdc with multicycles, either inside IP or when working on a design written by someone else. That being said, it is not the normal approach for designing with multicycles. Instead, a user will create logic and determine that the default setup and hold relationships are not what they want. They must determine what relationship they do want, and then apply multicycles to get that relationship.

The step of determining what relationship the user wants was left out, as it's not really a step for understanding TimeQuest, but a step in hardware design that could probably be a whole other chapter. Once the user determines what the new setup and hold relationships should be, they then use the steps above to determine what multicycle assignments would give that analysis.

I've added this comment because this question comes up for users who are new to multicycles and being taught how they work. In a training session, where there is no real design, it's easy to view this backwards. Remember, the normal flow is not, "I have multicycles in my .sdc and need to determine the new setup and hold relationships." Instead, it's usually "I've created hardware that needs a relationship different than the default. What multicycles do I need to apply to get the timing relationships to match my hardware?"

Multicycles - Two Common Cases

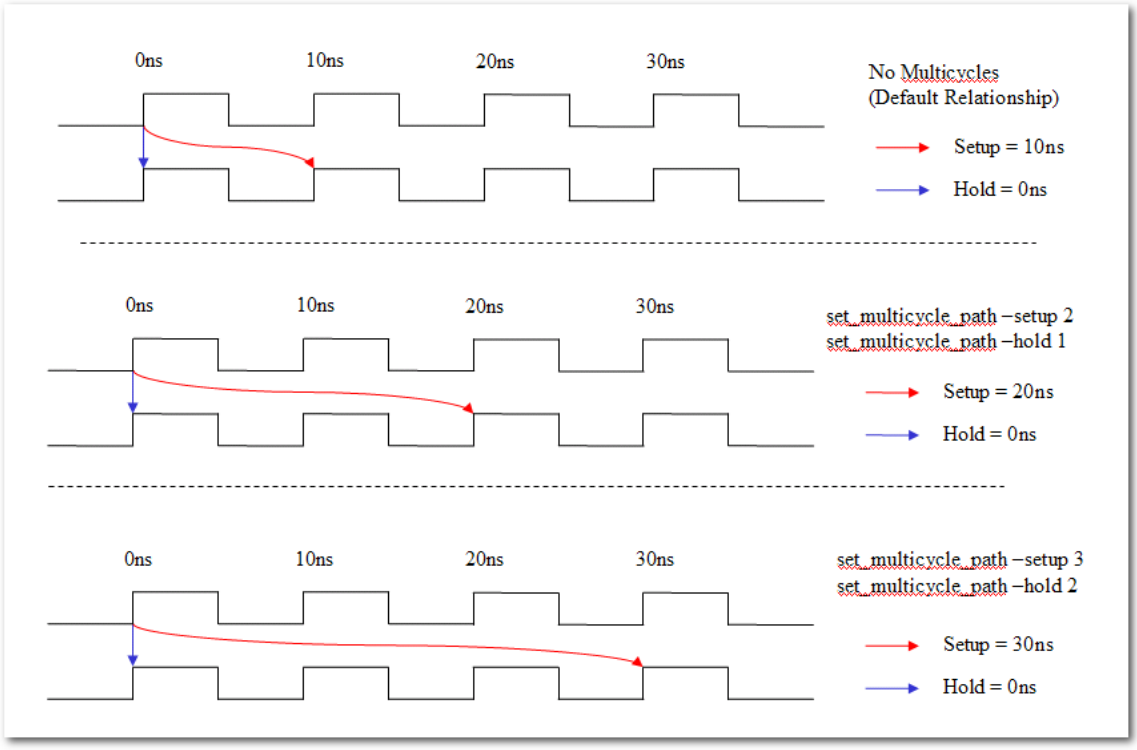
The previous section covered how multicycles affect setup and hold relationships, and will hold true for any clock relationships and any multicycle value. In reality, almost all multicycles fall under two different cases, and most users will be fine just understanding those two scenarios.

Case 1 - Opening the Window

When paths transfer data at a slower rate than the clock rate, users want to open the window. For example, let's say a design has a 10ns clock, but a group of registers in the design are fed by a toggling clock enable, and hence only toggle on every other clock. Since they are fed by a 10ns clock, the default analysis is a 10ns setup and 0ns hold, but the data is really transferring as if the clocks were 20ns, and hence a 20ns setup and 0ns hold is how the paths should be analyzed. The user wants to open the data window, making the setup relationship larger while keeping the hold relationship constant. This is done like so:

set_multicycle_path -setup -from src_reg -to dst_reg* 2*
set_multicycle_path -hold -from src_reg -to dst_reg* 1*

Note that the multicycle -hold assignment is necessary. Without it, the hold relationship would have been 10ns, which is not what the user wants. So to open the data window during analysis, the user needs to make their multicycle -setup with a value of N, and a multicycle -hold with a value of N-1. Here are two examples, where the user wants to open the data window to 2x and 3x its original size:



For an even larger data window, just continue the pattern. For example, if the user wanted to say the data could change anywhere between 0ns and 80ns, they would add:

set_multicycle_path -setup -from group_A -to group_B 8
set_multicycle_path -hold -from group_A -to group_B 7

I see many .sdc files filled with pairs of multicycles like this. Note that these multicycles are loosening the constraints, i.e making it easier to close timing. If the user knows a path runs at a lower rate while designing, it is recommended they make the multicycle constraints immediately, while the designer is intimately familiar with the logic. Too often designers say they'll do it later, and then spend time trying to find multicycle paths in their design to help close timing. Making the assignments up front is much easier.

Also note that these multicycles are often used for slow I/O interfaces. For example, when writing to an asynchronous RAM, the design might send out address and data, and then a few clock cycles later toggle the write enable signal. In this case, the address and data have extra

cycles to settle, and hence a multicycle to the I/O ports can help close timing. To give it three cycles, the designer might enter:

set_multicycle_path -setup -to [get_ports {SRAM_ADD[*] SRAM_DATA[*]} 3
set_multicycle_path -hold -to [get_ports {SRAM_ADD[*] SRAM_DATA[*]} 2

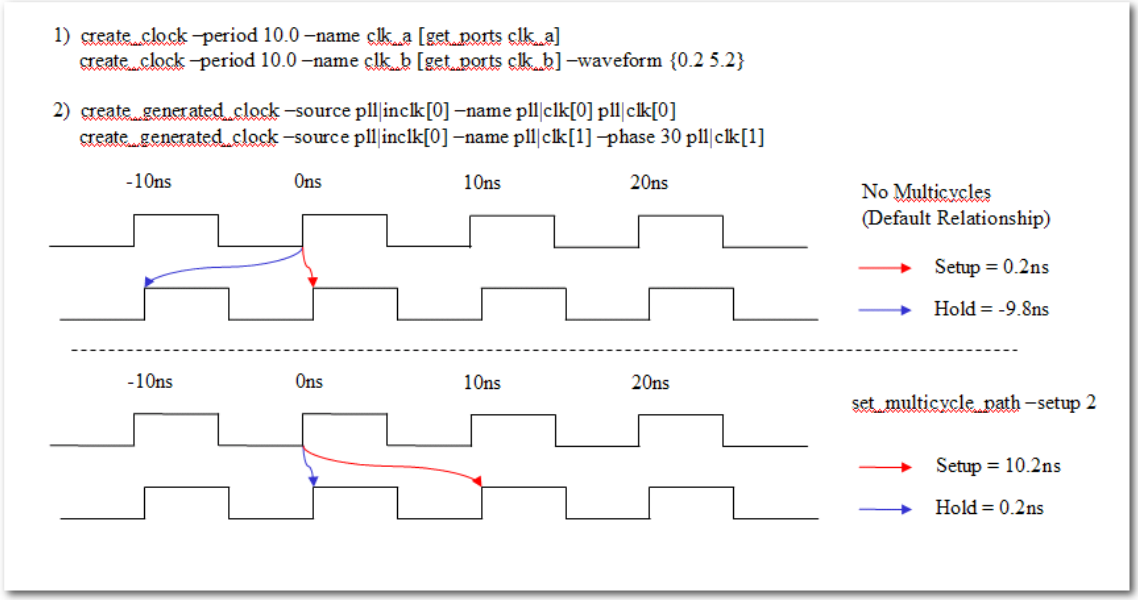
These multicycles assignments work in conjunction with the I/O constraints *set_input_delay* and *set_output_delay*, as described here.

Case 2 - Shifting the Window

This case occurs when a user's PLL does a small phase-shift on a clock, and that domain transfers data to/from other domains that do not have a phase-shift. For example, when the destination clock is phase-shifted forward and the source clock is not, the default setup relationship becomes that phase-shift. In the next example, the destination clock is phase-shifted by 200ps. I show two ways this could be done in the constraints. In example 1), clk_b is a base clock whose first rising edge is starts at 200ps rather than 0ps. The constraints in 2) are more common, whereby the PLL phase-shifts one of its outputs forward by a small amount. Both of these scenarios result in a default setup relationship of 0.2ns, which is pretty much impossible to meet, and probably not what the user intended.

If the user really wants data to transfer to the next edge, then they can add the following constraint to get the relationship shown in the second waveform:

set_multicycle_path -setup -from [get_clocks clk_a] -to [get_clocks clk_b] 2



There is no need for a multicycle hold, since the hold relationship follows the setup relationship. I call this "shifting the window", since the size of the data window between setup and hold is the same, it is just the next window that we're sending data through.

Note that the original relationship is not wrong, just not what the user intended. For example, if the destination clock were phase-shifted forward by a larger amount, say 5ns, then it's likely they would want the default relationship, which is a setup relationship of 5ns and a hold relationship of -5ns. It's only the small phase-shifts where user's often do not want the default relationship, but what constitutes a "small phase shift" must be decided by the user.

In recap, when adding a small phase-shift to a clock, a multicycle is often needed to shift the window that the data passes through. If the phase-shift is positive, they would add:

set_multicycle_path -setup -from [get_clocks base_clk] -to [get_clocks shifted_clk] 2

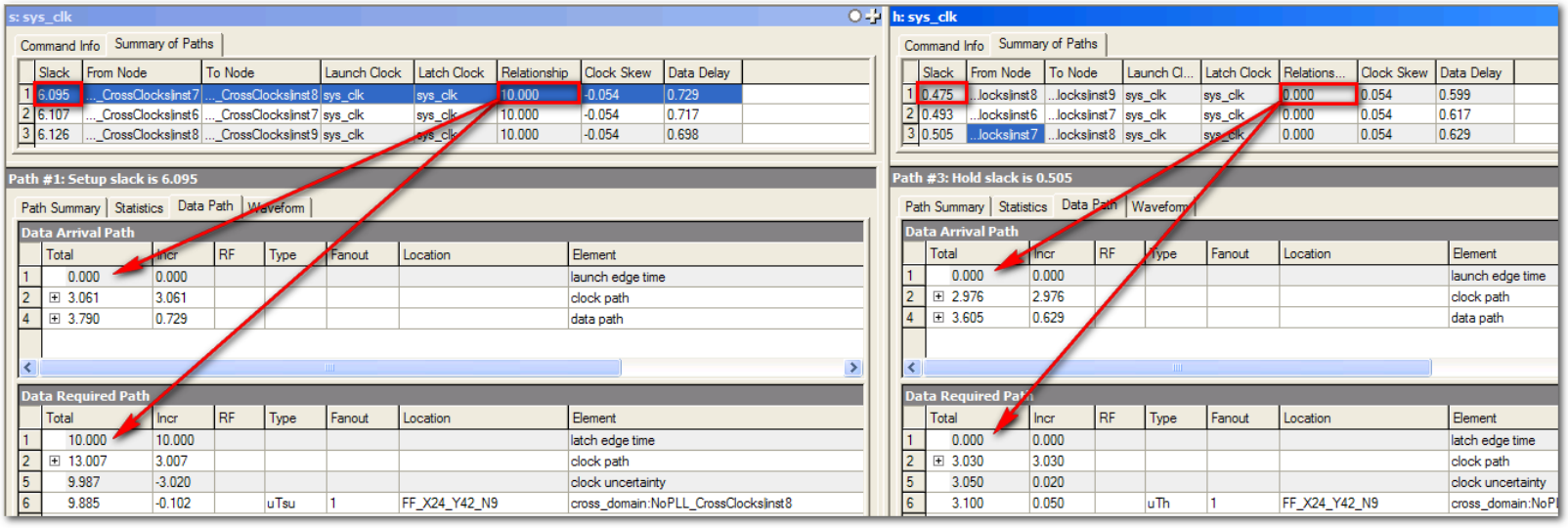
If the phase-shift is negative, then the constraint would be:

set_multicycle_path -setup -from [get_clocks shifted_clk] -to [get_clocks base_clk] 2

Of course, this only has to be applied where there are real clock transfers. If a design has forty clocks, and the user adds a small positive phase-shift on one of them, they do not have to add multicycles from the other thirty-nine clocks to this one. Most of the clocks will not have any paths to this shifted domain, and so the multicycle only needs to be applied between clocks with real connections.

Max and Min Delays

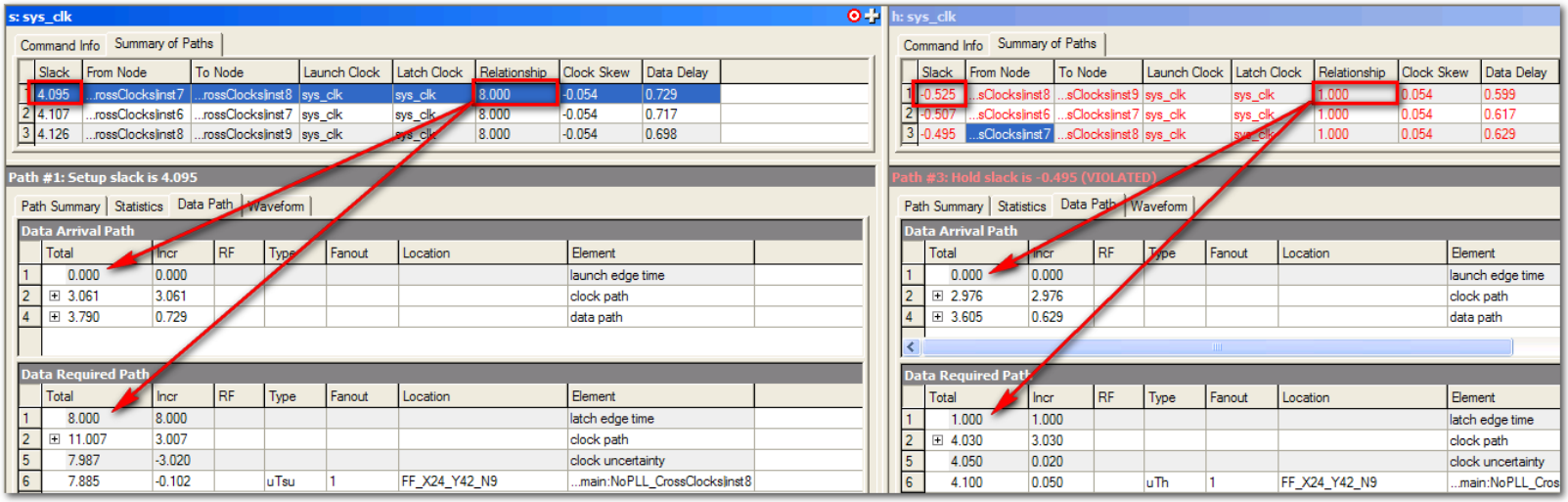
We have looked at calculating default setup and hold relationships and how to modify them with multicycles, which choose different clock edges of the existing waveforms. The constraints *set_max_delay* and *set_min_delay* allow users to modify setup and hold relationships to arbitrary values. In essence, these constraints are a low-level override allowing users to directly define setup and hold relationships. *Set_max_delay* directly modifies the setup relationship and *set_min_delay* directly modifies the hold relationship. Let's look at an example to understand it better. This first screen shot shows setup analysis on the left and hold analysis on the right within a 10ns clock domain:



As expected, the default setup relationship is 10ns, and the worst path meets timing with a slack of 6.095ns. The default hold relationship is 0ns, and the worst path meets timing with a slack of 0.475ns. If the user puts the following in their .sdc:

set_max_delay -from [get_clocks sys_clk] -to [get_clocks sys_clk] 8.0
set_min_delay -from [get_clocks sys_clk] -to [get_clocks sys_clk] 1.0

This example applies the constraint between clocks, so all paths between those clocks are modified, but in truth these constraints are more commonly applied between specific register or I/O endpoints. Re-running TimeQuest against the same fit shows:



Looking at the setup analysis on the left, the relationship has changed from 10ns to 8ns. That directly correlates to changes in the launch and latch edges. Note that everything else analyzed for this path is the same, i.e. the clock delay, data delay, etc. The slack has now changed from 6.095ns to 4.095ns, which is from the requirement being 2ns tighter. Likewise with the hold analysis on the right side, the hold relationship is now a positive 1ns, which changes the slack from +0.475ns to -0.525ns, and the design now fails timing.

An important note is that the physical clock delays are still being analyzed with this constraint, and hence clock skew still can affect whether or not a path meets timing. Users many times see the names *set_max_delay* and *set_min_delay* and assume it is constraining the data path independently of the clock paths.

The Dangers of set_max_delay and set_min_delay

The constraints *set_max_delay* and *set_min_delay* allow users to easily override default setup and hold relationships. It is important to note what is used to calculate the default setup and hold relationship to be aware of what information is being ignored when a user applies *set_max/min_delay* assignments. Some of the things that go into default analysis, such as clock period, make sense when they are overridden. For example, in the previous example we overrode a 10ns clock period and made it 8ns. The two places where I see problems are:

- Paths between registers clocked by different rise/fall edges

- Paths where one clock is phase-shifted or offset from the other

In theory, a user could use *set_max_delay* and *set_min_delay* to explicitly constrain their clocks. For example, if we wanted to overconstrain a 10ns clock to 8ns, we could apply:

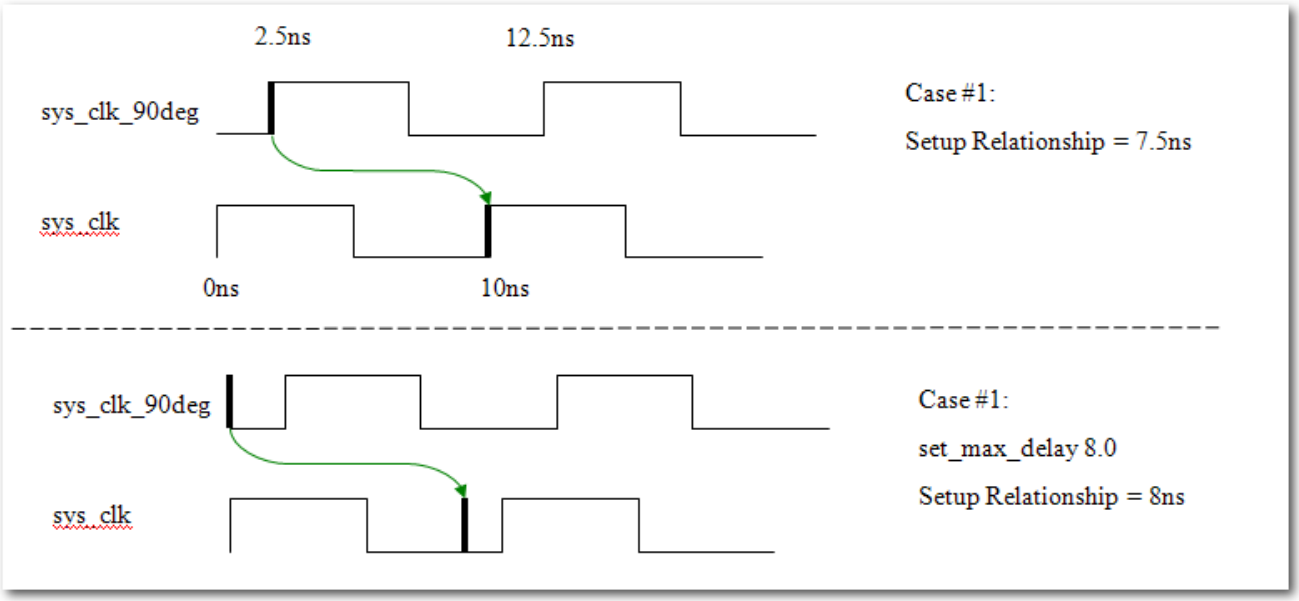
set_max_delay -from [get_clocks sys_clk] -to [get_clocks sys_clk] 8.0

This would overconstrain the setup relationship on every path in this domain from 10ns to 8ns while leaving the hold relationships at 0ns, exactly what we want. The problem is that clock relationships are more complicated, and clocks have multiple relationships, since falling and rising edges can be used. If sys_clk has any falling -> rising transfers or rising -> falling transfers, they would have a default setup relationship of 5ns. Our constraint has now loosened the requirement on those paths to 8ns. If we really wanted to overconstrain this domain, we would have to do:

set_max_delay -rise_from [get_clocks sys_clk] -rise_to [get_clocks sys_clk] 8.0
set_max_delay -fall_from [get_clocks sys_clk] -fall_to [get_clocks sys_clk] 8.0
set_max_delay -rise_from [get_clocks sys_clk] -fall_to [get_clocks sys_clk] 4.0
set_max_delay -fall_from [get_clocks sys_clk] -rise_to [get_clocks sys_clk] 4.0

This is necessary because the options -rise and -from are inclusive of all rise/fall edges, whereby we need to be more specific. Of course, if there are no registers clocked on the falling edge, none of this would have been necessary. Luckily, *set_max_delay* and *set_min_delay* are usually not used to constrain entire clock domains and are point solutions applied to specific paths, whereby the user knows whether rise and fall paths exist and constrains them accordingly.

The second issue involves phase-shifted clocks. Let's start with two 10ns clocks that are not phase-shifted. The default setup relationship is 10ns, and if wanted to overconstrain it by 2ns, we would apply a *set_max_delay* assignment of 8ns. Now let's say we phase-shift the launch clock forwared by 90 degrees.



The default setup relationship between these clocks is now 7.5ns. If we apply a *set_max_delay* constraint of 8ns, we've actually loosened the requirement. (Since *set_max_delay* and *set_min_delay* are independent of what the clocks look like, they are always analyzed with a launch edge of 0ns and a latch edge of whatever value is used in the assignment). In order to overconstrain this path by 2ns, we would want to enter a *set_max_delay* of 5.5ns. We have to take into account the phase-shift in our constraint.

Another way to think about this is to take an example where everything in the FPGA is constant except the phase-shift on the launch clock. With a 90 degree phase-shift, we have a 7.5ns requirement, and some slack number to tell us how much we meet timing by. If we change the phase-shift to 180 degrees, our setup relationship changes by 2.5ns, and our slack drops by 2.5ns. But if this path were constrained with a *set_max_delay* of 8ns, then the phase-shift on the launch clock could be 90 degree or 180 degrees and we would get the exact same slack. The phase-shift amount is ignored by *set_max_delay* and *set_min_delay* assignments.

This only applies if one clock is phase-shifted and the other is not. If both the launch and latch edges are phase-shifted 90 degrees, then that phase-shift would cancel out during analysis.

All of this may seem obvious, that phase-shifting the source clock by 90 degrees would reduce the default setup relationship to 7.5ns, and if I wanted to overconstrain this path I would need to enter a *set_max_delay* value less than 7.5ns. But when *set_max_delay* and *set_min_delay* assignments are used for I/O constraints, this is often missed.

Using set_max_delay and set_min_delay for Tsu, Th, Tco, Min Tco and Tpd

The constraints Tsu, Th, Tco and Tpd are called device-centric constraints, as they constrain the I/O ports of the FPGA device independently of its environment. These constraints are not directly supported by TimeQuest, as it uses the system-centric constraints *set_input_delay* and *set_output_delay*. They are called system-centric, since they will change due to changes in system requirements, such as a change in the clock period or board delays to external devices.

The Classic Timing Analyzer(the original static timing analysis engine in Quartus) used device centric-constraints, and so when TimeQuest was first released, all Quartus user's were accustomed to using Tsu, Th, Tco, Min Tco and Tpd assignments. They had been doing this for years and as a result did not like moving to *set_input_delay* and *set_output_delay* assignments, which are reallythe ones designed for constraining I/O. As a result, Altera showed users how to use *set_max_delay* and *set_min_delay* as substitutes for these device-centric constraints. Equations were given comparing the two:

Input Ports:
set_max_delay -from [get_ports {<input>}] <Tsu_Requirement>
set_min_delay -from [get_ports {< input >}] -<Th_Requirement>

Output Ports:
set_max_delay -to [get_ports {<portname>}] <Tco_Requirement>
set_min_delay -to [get_ports {<output>}] <MinTco_Requirement>

Combinatorial Paths through Device:
set_max_delay -from [get_ports {<input>} -to [get_ports {<output>}] <Tpd_Requirement>

set_min_delay -from [get_ports {<input>} -to [get_ports {<output>}] <minTpd_Requirement>

Note that the <Th_Requirement> is negated. Everything else is directly converted.

So if a user wanted to constrain a 32-bit input bus call ram_data as well as a bit called ram_parity to a Tsu of 4ns and Th of 1ns, they would use the equation and write:

set_max_delay -from [get_ports {ram_data[*] ram_parity}] 4.0
set_min_delay -from [get_ports {ram_data[*] ram_parity}] -1.0

This method was especially useful when converting legacy designs with Tsu/Th/Tco type constraints over to TimeQuest. It's important to understand what is going on though.

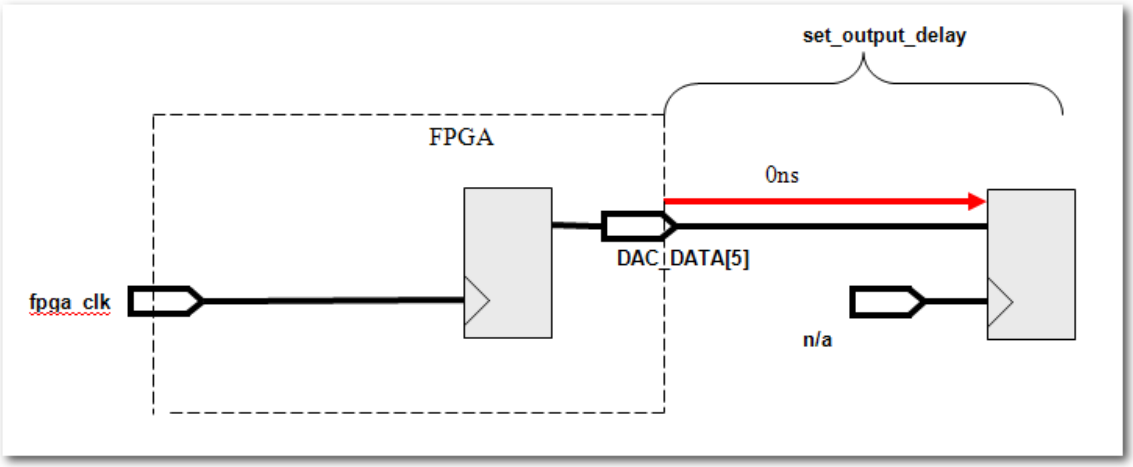
The constraints *set_max_delay* and *set_min_delay* were not designed to be I/O constraints. As discussed, they are low-level constraints that override the default setup and hold relationship. But how do they work on an I/O port that doesn't have a default setup or hold relationship? The answer is that TimeQuest infers a *set_input_delay* or *set_output_delay* constraint. For example, if I constrain an output port like so:

set_max_delay -to [get_ports dout] 5.0
set_min_delay -to [get_ports dout] 1.0

TimeQuest will infer the following:

set_output_delay -max -clock n/a 0.0 [get_ports dout]
set_output_delay -min -clock n/a 0.0 [get_ports dout]

These inferred *set_output_delay* assignments state that port dout drives an external register that is clocked by clock n/a, and the delay to that register is exactly 0ns. It looks like so:



The fpga_clk is the clock coming into the FPGA and constrained by the user. If it goes through a PLL or gated clock, those must be constrained too.

The clock n/a stands for "Not Applicable" since it's not a clock that has been defined. It doesn't matter what this clock looks like, since it's setup and hold relationships to the fpga_clk get overridden by the *set_max_delay* and *set_min_delay* assignments(which is how these

assignments work, as described). Since the external delay is 0ns, it has no affect on the final slack calculation. What is left is that our assignments override the setup and hold relationships between fpga_clk and n/a clock to make them 5ns and 1ns. Since the external delays are all 0ns, then the FPGA's delays are the only thing used, and must meet the 5ns setup relationship and 1ns hold relationship.

The key to this is that it looks like the user has done the constraint with TimeQuest's normal constraints, *set_input_delay* and *set_output_delay*. The I/O ports now have a full register to register path, and can be reported as normal setup and hold paths. If the user runs:

r*eport_timing -setup -detail full_path -to_clock n/a -npaths 200 -panel_name "Tcos"*
r*eport_timing -hold -detail full_path -to_clock n/a -npaths 200 -panel_name "min Tcos"*

They will see all setup and hold analysis to these external registers clocked by clock "n/a". Looking at one of the setup paths:

Report Timing

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	1.480	domain:inst6|inst4	dout	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	n/a	5.000	-0.464	3.056

Path #1: Setup slack is 1.480

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 0.464	0.464					clock path
3	0.000	0.000					source latency
4	0.000	0.000			1	PIN_N3	sys_clk
5	0.000	0.000	RR	IC	1	IOIBUF_X53_Y22_N1	sys_clk~input|i
6	0.753	0.753	RR	CELL	2	IOIBUF_X53_Y22_N1	sys_clk~input|o
7	3.001	2.248	RR	IC	1	PLL_R2	the_system_pll|altpll_component|auto_generated|pll1|inclk[0]
8	-3.226	-6.227	RR	COMP	3	PLL_R2	the_system_pll|altpll_component|auto_generated|pll1|observablevcoout
9	-3.226	0.000	RR	CELL	1	PLL_R2	the_system_pll|altpll_component|auto_generated|pll1|clk[1]
10	-1.772	1.454	RR	IC	1	CLKCTRL_G10	the_system_pll|altpll_component|auto_generated|clk[1]~clkctrl|inclk
11	-1.625	0.147	RR	CELL	15	CLKCTRL_G10	the_system_pll|altpll_component|auto_generated|clk[1]~clkctrl|outclk
12	0.082	1.707	RR	IC	1	FF_X29_Y1_N9	inst6|inst4|clk
13	0.464	0.382	RR	CELL	1	FF_X29_Y1_N9	domain:inst6|inst4
14	⊟ 3.520	3.056					data path
15	0.564	0.100		uTco	1	FF_X29_Y1_N9	domain:inst6|inst4
16	0.564	0.000	RR	CELL	1	FF_X29_Y1_N9	inst6|inst4|q
17	1.085	0.521	RR	IC	1	IOOBUF_X29_Y0_N95	dout~output|i
18	3.520	2.435	RR	CELL	1	IOOBUF_X29_Y0_N95	dout~output|o
19	3.520	0.000	RR	CELL	0	PIN_Y11	dout

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	5.000	5.000					latch edge time
2	⊟ 5.000	0.000					clock path
3	5.000	0.000	R				clock network delay
4	5.000	0.000	R	oExt	0	PIN_Y11	dout

The setup relationship is 5ns, which is the value entered for the *set_max_delay* assignment. Data Required Time uses all the delays through the FPGA, which can be seen going down the Location column, from the clock entering on Pin_N3, through the IO buffer and PLL, the global clock tree G10, the output FF, IO Output Buffer and finally Pin Y11. This all takes 3.520ns. The Data Required Path is outside the FPGA, and just has the latch edge time of 5ns, which is our requirement. The external delay oExt is 0ns, which means the external delay has no affect. So our requirement is 5ns, our delay through the FPGA is 3.520ns, and we meet timing by 1.480ns. Another way of stating this is that the Tco is 3.520ns and meets our requirement by 1.480ns. In this case, using *set_max_delay 5.0* to our output port analyzed the path the same way as a Tco requirement of 5ns would have. Everything looks good.

But there are two dangers with *set_max_delay* and *set_min_delay* assignments. The first issue is that negative edge analysis is ignored. This is not a big deal since Tsu, Th, and Tco work that way too. A good example I often use it to think of a 20ns clock driving an output register, where the Tco is reported to be 7ns. If the user modifies their code so the output register clocks on the falling edge. In hardware, the output signal is now changing a half period, 10ns, differently than it was before. How should the Tco value be reported? Does it change by a half period to -3ns, to 17ns, or stay at the old value of 7ns? The answer is that it does not change at all and would still be reported as a 7ns Tco. This was always a problem with device-centric constraints. In this example, two output ports could have identical Tcos, but if one were clocked on the rising edge and the other on the falling edge, their data would come out at very different times. The constraints *set_max_delay* and *set_min_delay* will also ignore falling edges, since the falling edge clocks affect the setup and hold relationships, and these constraints override those relationships. This one is not that big of a deal though, since *set_max_delay* and *set_min_delay* are behaving the same way as device-centric constraints Tsu, Th, and Tco.

The second danger is unexpected. Phase-shifts are also ignored by *set_max_delay* and *set_min_delay* constraints. In the report_timing diagram above, the clock in the FPGA goes through a PLL. If that PLL has a manual phase-shift, it would show up in the launch and latch edges, but these are overridden by the *set_max_delay* and *set_min_delay* requirements, and are basically ignored. The PLL could do no phase-shift, a 90 degree phase-shift, a -270 degree phase-shift, i.e. anything, and the analysis would be the same. The slack would be identical. This is exactly how *set_max_delay* and *set_min_delay* are supposed to work, but it is not expected for I/O constraints.

Most likely, the user needs to compensate for phase-shifts in their constraints. For example, if the PLL did a phase-shift of 0.2ns, then the data will come out 0.2ns later. To compensate for that, the user would need to make their *set_max_delay* assignment 0.2ns tighter, at 4.8ns, and their *set_min_delay* assignment 0.2ns looser, at -0.2ns. The concerning part is that there is no warning or anything that the PLL phase-shift is being ignored, as the *set_max_delay* and *set_min_delay* assignments are working as they're supposed to. This is one of the main reasons I recommend against using these constraints as a substitute for device-centric I/O constraints. (There are others. Look at Device-Centric and System-Centric constraints.) Even for users that understand this, I consider it dangerous in a project. The project may eventually get passed to anther engineer, who begins modifying the phase-shift of clocks coming out of the PLL but doesn't know to modify their I/O *set_max_delay* and *set_min_delay* constraints accordingly.

In summary, the big concern with using *set_max_delay* and *set_min_delay* is that any manual PLL phase-shift is not used in calculating slacks. As long as there is no phase-shift to the I/O registers, it works quite nicely, and can still be used if the customer "accounts for phase-shifts" in their requirements. The .sdc could always do something like:

set phase_shift 0.0 ;#PLL phase shift in nanoseconds

#Input Ports:
set_max_delay -from [get_ports {<input>}] [expr <Tsu_Requirement> + $phase_shift]
set_min_delay -from [get_ports {< input >}] [expr -<Th_Requirement> + $phase_shift]

#Output Ports:
set_max_delay -to [get_ports {<portname>}][expr <Tco_Requirement> - $phase_shift]

set_min_delay -to [get_ports {<output>}] [expr <MinTco_Requirement> - $phase_shift]

This method creates a variable called phase_shift and uses it in all the device-centric I/O constraints. The user must remember to update this variable if they modify the PLL phase-shift, as there is nothing that will warn if it is incorrect.

I personally think it is much better to learn how to use *set_input_delay* and *set_output_delay* constraints, as they are not very difficult and are meant for I/O constraints. I wanted to explain this danger for those who choose not to.

A few other notes:

- This explains why a clock called "n/a" shows up. It will be the Launch Clock for input ports and Latch Clock for output ports.

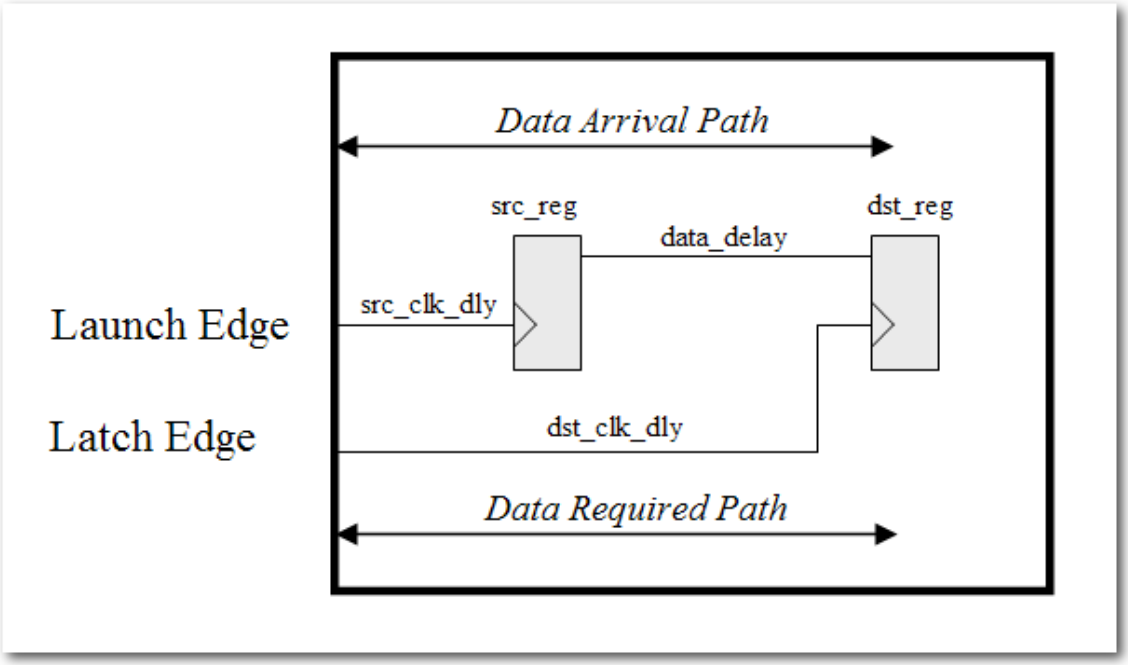
- Now that an external register exists, the I/O paths can be reported as register-to-register transfers. So input registers are fed from an external register clocked by n/a, and outputs feed an external register clocked by n/a. To analyze these paths, you can still use the ports though, such as:

report_timing -setup -from [get_ports {<input list>}] -npaths 100 -detail full_path
-panel_name "Tsu"
report_timing -hold -from [get_ports {<input list>}] -npaths 100 -detail full_path
-panel_name "Th"
report_timing -setup -to [get_ports {<output list>}] -npaths 100 -detail full_path
-panel_name "Tco"
report_timing -hold -to [get_ports {<output list>}] -npaths 100 -detail full_path
-panel_name "minTco"

- The inference of *set_input_delay* and *set_output_delay* only occurs if the user does not have their own external delay constraint. If the designer already has this constraint on the I/O, then the values from that constraint are used. Note that *set_input/output_delay* and *set_max/min_delay* constraints do not compete with each other, and instead work together.

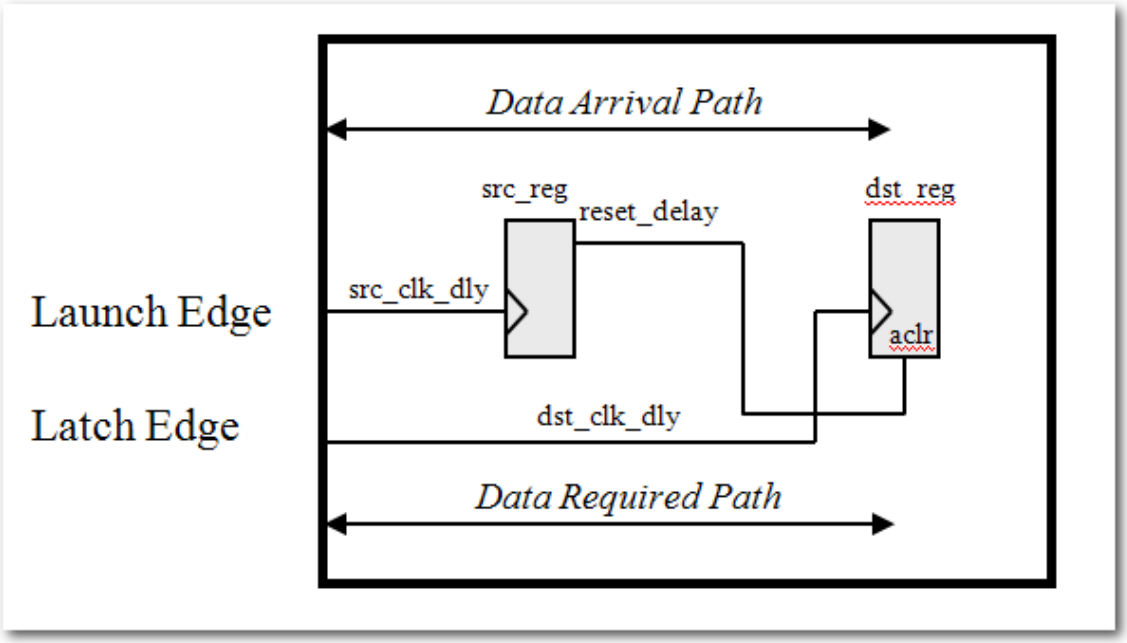
Recovery and Removal

Recovery and Removal analysis is one of those things where what is occurring is very easy to understand, but the why is very difficult. As a recap, the user describes clock waveforms in their SDC file, and from there TimeQuest determines setup and hold relationships. The basic principle is that the launch edge clocks data from the source register, and it must get to the destination register before the setup latch edge and after the hold latch edge. This is described in detail at the beginning of this section. Note for setup and hold we drew the following diagram:



The data_delay path feeds a synchronous port on dst_reg, whether it be the data input, the clock enable, the synchronous clear, etc. It is anything that is clocked in by the latch clock.

Recovery and removal is an identical analysis, except data_delay feeds an asynchronous input on dst_reg. Here's the new schematic, where the only differences are that an asynchronous port on the destination register is being driven, and the name of signal was change to reset_delay.

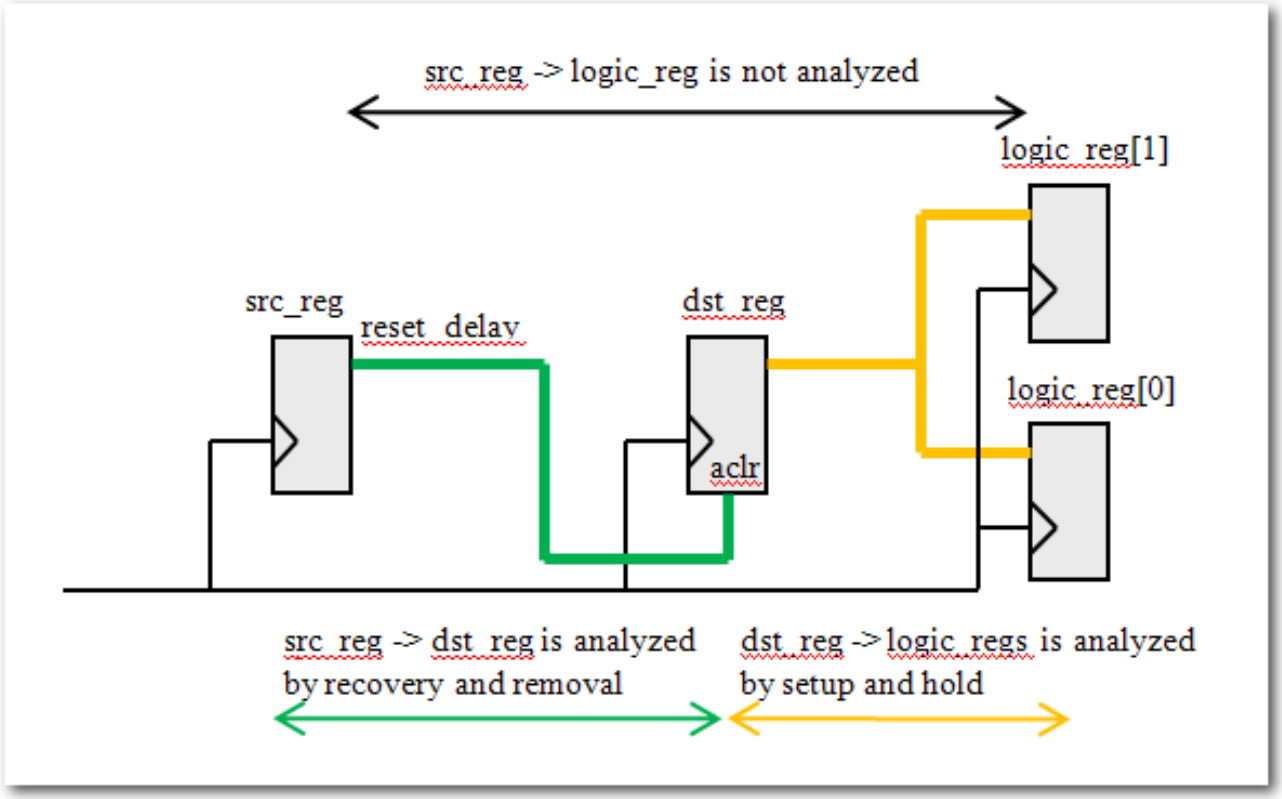


For simple understanding, if these registers were in a 10ns clock domain, and ignoring clock skew, recovery states that the reset_delay must be less than 10ns, and removal states that the reset_delay must be greater than 0ns. Recovery analysis is identical to setup analysis, except the signal feeds an asynchronous port on the destination register. Removal analysis is identical

to hold analysis, except the signal feeds an asynchronous port on the destination register. To my understanding, some tools don't even have "recovery and removal", they just analyze all transfers, whether they're synchronous or asynchronous, as setup and hold.

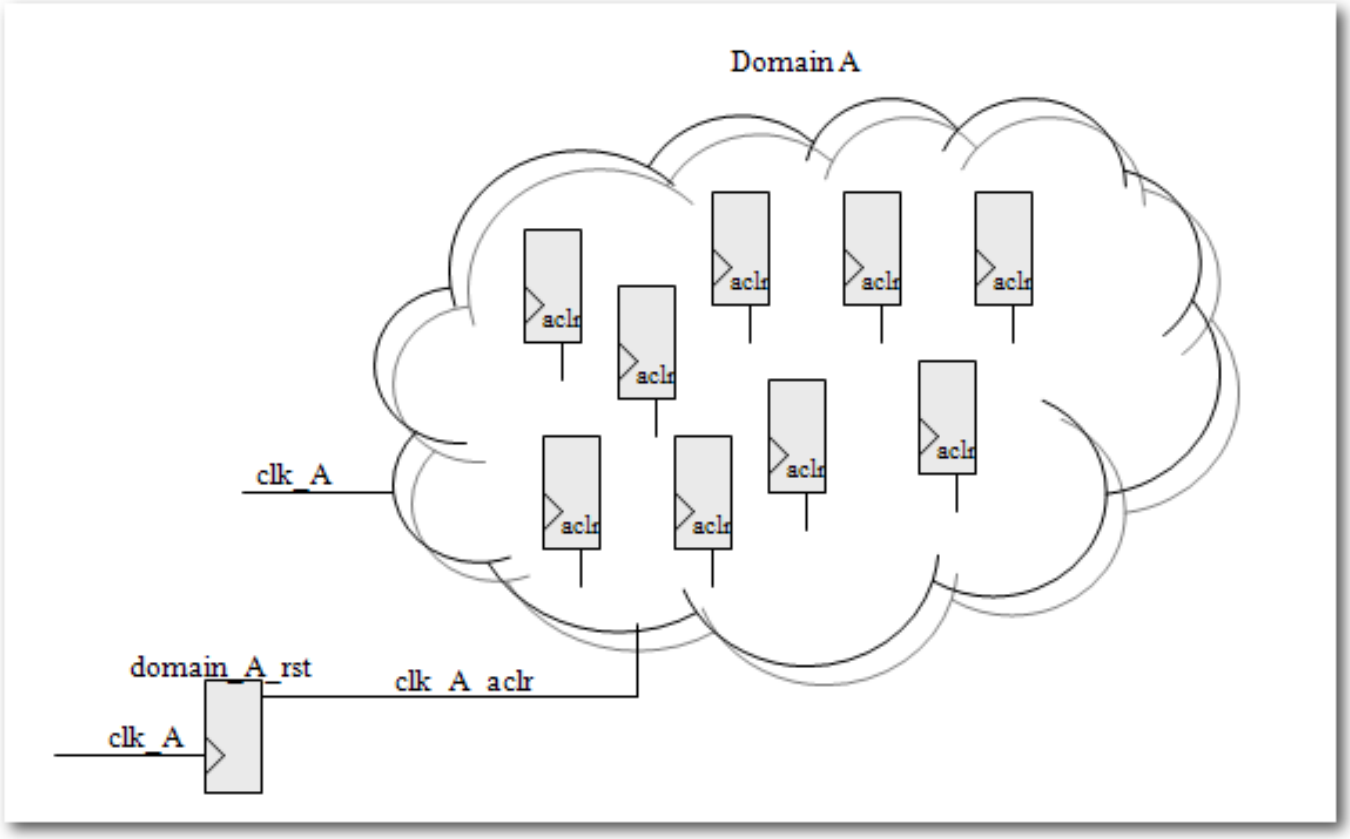
With that understanding, EVERYTHING else about setup and hold relates to recovery and removal. Default relationships are determined the same way. Multicycles have the same affect(*set_multicycle_path -setup* will affect the recovery analysis, and *set_multicycle_path -hold* will affect the removal analysis). The constraints *set_max_delay* and *set_min_delay* still act as low-level overrides of the relationships, and *set_false_path* will still cut timing analysis on a reset path.

One thing I do want to make clear is the asynchronous register is an endpoint. There is no analysis through an asynchronous port. In the diagram below, the recovery/removal analysis is only from src_reg to dst_reg, and separately dst_reg to the logic_regs are analyzed as setup and hold:



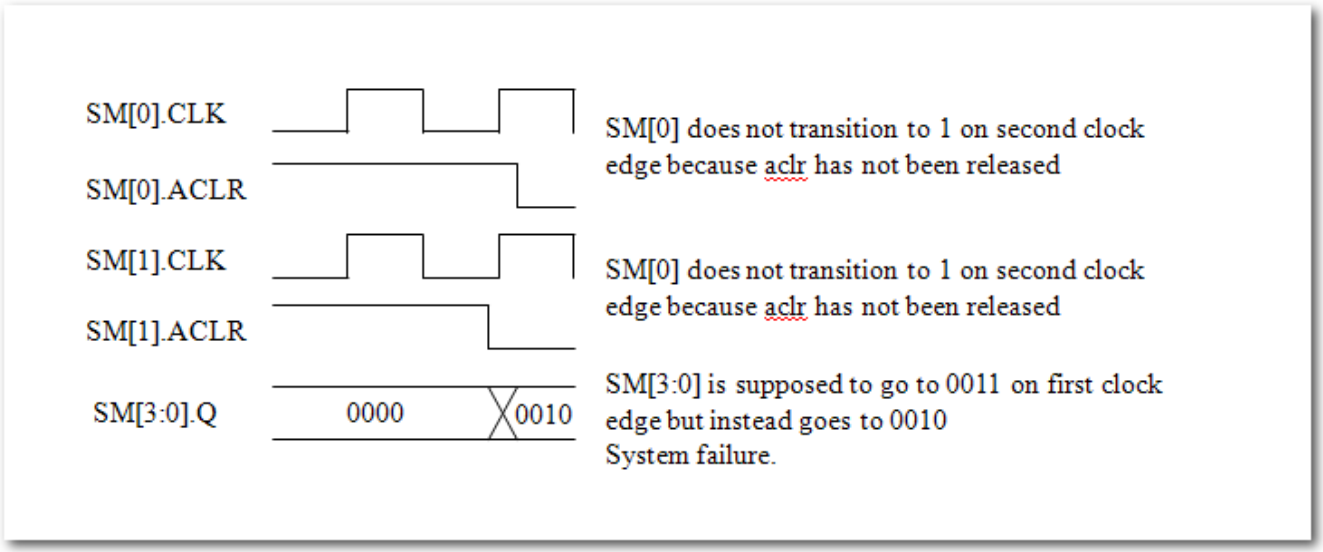
There is absolutely no analysis from src_reg through dst_reg to the logic_regs. As a result, when src_reg resets dst_reg we know it will occur within a clock cycle, but the asynchronous change in dst_reg might reach the logic_regs before the next latch edge, after it, some combination of both(where one logic_reg sees the new value and the other does not), or right on the latch clock edge, causing logic_reg to go metastable. The moral is that the user should NEVER use asynchronous ports for general logic and only use them as a domain-wide reset. If the logic_regs are also asynchronously reset by src_reg, then the analysis from src_reg through dst_reg to the logic_regs does not matter, since the logic_regs are also being reset and hence immune to changes on their synchronous inputs.

If you understand the fundamentals of setup and hold analysis then you understand the fundamentals of recovery and removal. The difficulty is usually not in understanding what is being analyzed buy why. Let's look at an example where a register asynchronously resets an entire domain:



In this schematic, the register domain_A_rst fans out to the aclr port of all the other registers in Domain A. There might be 10 registers or 100,000 registers, it doesn't matter Note that the resets source and destination registers are synchronous to each other.

The best way to explain why recovery and removal is analyzed it to show a failure. So let's pretend domain_A_rst is clocked asynchronously to Domain A, or that clk_A_aclr is not analyzed by recovery and removal. Either way, the point is that we have no analysis on when clk_A_aclr feeds the registers in Domain A. Let's look at a 4-bit binary state-machine within Domain A that resets to state "0000" and on the first clock cycle is supposed to transition to state "0011".



Due to the aclr port not being timed, it reaches the 4 bits of SM at different times that the user cannot analyze. On one release from reset, it get to SM[1] before SM[0] and a clock edge occurs in between. This releases bit SM[1] on the second clock edge, but still hold SM[0] in the reset state, so only some bits of the state-machine transition. This could cause it to enter an unknown state, or possible a known state that will never be valid because earlier parts of the state-machine were never reached. This could cause a system failure, and is exactly what recovery and removal prevents. By ensuring all registers within a domain are released from reset on the same latch edge, this type of failure is avoided, and the system comes out of reset the same way, every time.

In general, logic that changes on the first clock out of reset and that can hold its state is the most susceptible to recovery/removal failures. (Logic that doesn't hold its state, like a simple multiplier, may calculate the incorrect value out of reset, but that value usually filters out of the device before anything is done with the incorrect value). Because of this, most logic is immune to recovery/removal failures. The problem is that there is no tool to determine which logic is immune and which is not. Also, recovery/removal cannot be simulated since there are too many different combinations of how the registers could come out of reset. Finally, recovery and removal failures are extremely difficult to debug in the lab since they usually only occur on a small percentage of system resets. Who hasn't had a design not work in the lab, the designer does a reset, and everything starts working? These problems are usually ignored, but may show up as periodic failures in the field.

I have received the frantic calls from designers who have designs in the field exhibiting a particular failure rate out of so many power-ups/resets. This is not the time to find out about a timing issue like this, and my general suggestion is to design proper resets up front, close timing on them, and get it right from the beginning.

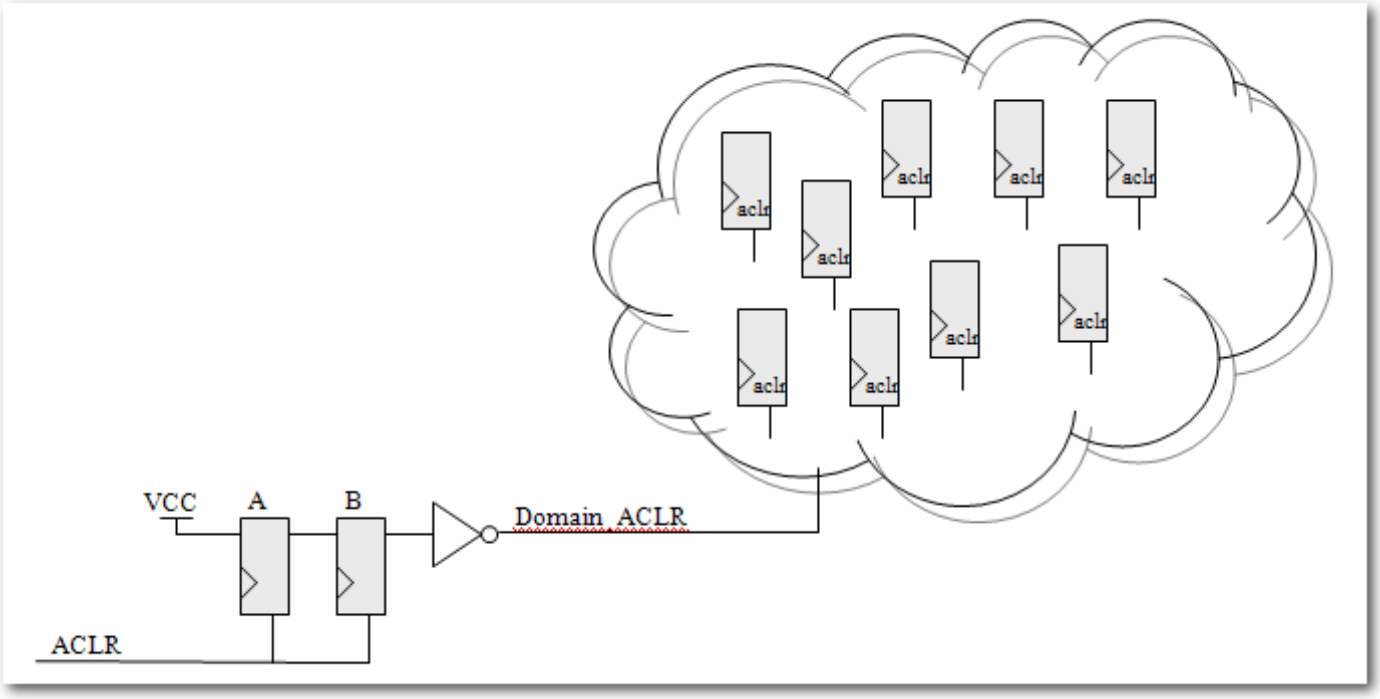
Once designers understand what a recovery/removal failure looks like, the inevitable follow-up question is, "Why not make the reset synchronous?"

There are two reasons for this. The first is that many designs require an asynchronous reset for reliability. The great benefit of an asynchronous reset is that it can be reset without a clock. So if the system failure occurs by a clock generator going bad, the user can still reset the design. If it's a medical scanning device that is emitting particles at the patient, the

asynchronous reset allows them to reset to a safe state. If the device controls an automobile and the user is accelerating when the clock driver goes bad, they can still reset to a non-accelerating state. For conditions like these, it is imperative the reset be asynchronous.

Of course many designs don't care. If the clock driver fails on a handheld video game system, most likely the whole device is getting thrown out. Minimally, the user doesn't care what happens during this type of failure, and has no safety/security requirements. But the second reason for using an asynchronous reset is that it gets better results in Altera FPGAs. There is a dedicated asynchronous set/reset on each register, and if the design doesn't use them, they are wasted. More importantly, if the reset is synchronous, it will use up synchronous inputs, whether it be the synchronous clear port or an input to the LUT, that could have been used for general logic. So making the domain-wide reset synchronous will make the design a little larger and a little slower.

We've said that the reset needs to be asynchronous for resetting the logic in case the clock driver fails, and it has to be synchronous for de-asserting so that all registers can be timed and come out of reset on the same cycle. So is the reset synchronous or asynchronous? The answer is a circuit called the "asynchronous assert, synchronous de-assert reset" and looks like so:



As can be seen, when ACLR asserts, it asynchronously goes through register B and sets/resets all the registers in the domain without the need for any clock edges. This satisfies the requirement to asynchronous assert reset. But when it de-asserts, there must be two clock cycles for the VCC to travel through registers A and B before synchronously de-asserting all the logic in the domain. This satisfies the requirement that the de-assert can be timed synchronously. Two registers are used for metastability, since ACLR is often released asynchronously to the clock. As the name says, it is an asynchronous assert, synchronous de-assert reset. Now TimeQuests's recovery and removal analysis can properly time everything from register B to all the registers in the domain.

My feeling is that without good reason, this structure should be in every design. Some quick notes on it:

- ACLR can be completely asynchronous to the domain in question.
- The ACLR signal is usually more complex. Besides some sort of user logic to determine when to reset(whether it be a push-button, a Nios command, an external CPU, etc.), there are usually conditions for coming out of reset. The most common is making sure the PLL for that domain has asserted its lock signal.
- The asynchronous assert, synchronous de-assert should be repeated for every group of clocks. A group consists of all related clocks. For example, if a PLL creates a 50MHz, 100MHz and 200Mhz clock that are all related, only one circuit is necessary to reset all three domains. (There is no reason the user can't create multiple versions). In general, the two registers should be clocked off the slowest clock domain. This provides the easiest timing requirements to the slower domains and provides consistent releasing of each clock domain.
- For timing closure of recovery and removal, the Domain_ACLR net often meets timing by being put on a global. If globals are available and the domain is relatively large, that's the best option. Sometimes a domain is too fast for a global though. For example, the clock tree may be longer than 4ns, which is too slow if the domain has a 4ns period and hence recovery needs to meet that requirement. In these cases, taking the net off of a global often gives better timing.
- If timing closure is still difficult in fast domains, the structure can be repeated for various hierarchies within the domain, reducing the fan-out and distance the net must drive.
- Recovery and Removal failures can hurt setup and hold timing. The fitter considers recovery and removal timing to be just as important as setup and hold, and will try to balance timing between the two if it achieves a better overall slack. This is especially true when the Domain_ACLR(see above schematic) net is not on a global and has a tight requirement. This will pull all the destination logic close to register B, which could be at the expense of timing within the domain. I recommend trying to close timing on recovery and removal early on, since it is usually not very difficult. Another option is to add a temporary:

set_false_path -from B

B would naturally be replaced by the name of the register driving the asynchronous set/reset of the domain. Since recovery/removal failures occur out of reset and are usually very sporadic, it's easy to work around them in the lab. If this helps meeting setup timing, it would allow the designer to continue debugging the rest of their logic. The designer must be careful to later remove the false path and fix the recovery timing issues.

I've found recovery and removal to get a varied reception. ASIC designers are usually fully aware of this topic(even if they didn't know the terms recovery and removal) and are already building reset structures to meet their needs. On the other hand, many FPGA designers have never paid attention to it, which was not helped by the fact that the Classic Timing Analyzer did not do Recovery/Removal analysis by default, and the user had to go deep into the menus to turn it on. These designers often ignore this whole topic and pretend it's not an issue. There does seem to be a general shift though, from ignoring it, to being aware, to understanding its importance.

Section 3: SDC Constraints

This section discusses the major SDC constraints. It is not meant to re-state the basics, but as an additive source of information. To help understand a command, type *command_name -long_help* in TimeQuest, e.g. to learn more about *set_multicycle_path*, type:

set_multicycle_path -long_help

To get a list of available commands, some suggested commands to type:

help
help sdc
help sdc_ext

The basic syntax for commands is to have the command followed by a list of options that have a description, such as *-period 10.0*. Some commands have required options that do not have a descriptor. For example, *set_output_delay* has two options <value> and <target>. The <value> is the numerical delay for this command, and the <target> is what port/s it is applied to. This can be confusing at first, as an SDC might have:

create_clock -period 10.0 -name sys_clk sys_clk

As can be seen, sys_clk is listed twice. The first one is linked with the -name option, and is the user's name given to the clock. The second is the <target> it is applied to, which is a port in the design called sys_clk. It could also be written like so:

create_clock -period 10.0 -name sys_clk [get_ports sys_clk]

The square brackets execute a command inside and return a value, so the command *get_ports* finds any port that matches sys_clk and returns it.

create_clock

Note: In TimeQuest, type "create_clock -long_help" for more information.

This constraint is used to create base clocks. There are two major uses, the first being to create a clock constraint coming into the FPGA. The second major use is when the constraint does not have a <target> and is a virtual clock, which is used for I/O analysis. The launch and latch edges for these clocks start at the commands target, which is why it is not recommend to apply this constraint to clocks inside the FPGA. For example, if the user has a divide-by-2 register in their design, they might do something like so:

create_clock -period 20.0 -name div_clk [get_registers clk_blk:u2|div2reg]

The problem with this is that the delay to div2reg is not analyzed, so on one fit it might be 4ns and the next fit it is 8ns, while in both cases that delay is ignored. That is why this is only recommended for clocks coming into the FPGA or virtual clocks outside of the FPGA. Also note that *set_clock_latency* can be used in conjunction with this constraint to account for external delays.

Options:

-*waveform* - This is used to describe what the clock looks like. If not used, the clock defaults to having a rising edge at 0ns, a falling edge at (period/2), and repeated edges from there. I recommend using -waveform only if the user does not want this default, since it is a possible source of error when changing a clock period. For example, if a design has:

c*reate_clock -period 10.0 -name sys_clk -waveform {0 5} [get_ports sys_clk]*

That constraint is correct, but if the period ever changes to 8ns, it is easy to just modify the -period option and leave -*waveform {0 5}*. I have seen this done. The design now has an 8ns clock with a rising edge at 0ns and a falling edge at 5ns, which is no longer a 50% duty cycle.

The -waveform also allows for a clock offset. For example, if a clock has a -period 10.0, a 2ns offset could be specified with -*waveform {2.0 7.0}*. The clock still has a 50/50 duty cycle, but is offset by 2ns. Note that there is no way to do a negative offset since the first rising edge must be between 0ns and the period, but they can shift it a whole cycle to accomplish the same thing. For example, if the user wanted to represent a -2ns offset on a 10ns clock, they would add -*waveform {8 13}*. Due to periodicity, TimeQuest's default setup and hold relationships work out the same way.

-*add* - If two *create_clock* assignments are applied to the same target, the second assignment will be ignored and a warning will be issued. This option on the second assignment means that it describes a second clock coming into the device. An example where this is used is if a device plugs into two different boards, and the legacy board might drive a slower clock into the FPGA. This allows TimeQuest to analyze both scenarios.

create_generated_clock

Note: In TimeQuest, type "create_generated_clock -long_help" for more information.

This command is used to create clocks based off of other clocks. The most common uses are:

- PLL outputs. These are generally covered by *derive_pll_clocks*.
- Source synchronous outputs. This constraint is applied to the port sending a clock off chip, and then used as the -clock option on the data's *set_output_delay* constraint.
- Clock muxes. Although not always necessary, generated clock assignments on the output of a clock mux, based on the clocks coming into the mux, give the user flexibility in analyzing and constraining the muxed domains.

- Ripple clocks. Any time the output of a register feeds the clock port of another rgister, that is a ripple clock. The source register requires a generated clock assignment or else the ripple clock will be unconstrained.

Note that it is acceptable to have generated clocks of generated clocks. I had a design with a *create_clock* on the clock coming in, which then went through a PLL, a ripple clock, a clock mux, and then fed out as a source synchronous output. There were four generated clock assignments, at the PLL output, the ripple clock, the mux and the output, each based on the previous clock. We were able to correctly constrain and analyze timing through the whole design.

Options:

-name - The name of the newly generated clock.

-divide_by/-multiple_by - Used to divide and/or multiply the incoming clock period.

-phase/-offset - Used to shift the clock edges from the incoming clock. A PLL is the only thing that can really shift a clock, so that is really the only place this would be used. In the end, these two options can do the same thing, and really depend on how the user wants to represent the shift, since -phase is based on the incoming clock period and -offset is a fixed time delay.

-invert - This is used when a clock is inverted and TimeQuest does not recognize the inversion. The only time I use this is if I'm sending a clock off-chip through an altddio_out megafunction, and I tie the high register's input to GND and the low register's input to VCC. This inverts the clock as it leaves the FPGA, but in a way that TimeQuest does not recognize, and hence the -invert option is necessary. When a user inserts an inversion on their clock line in RTL, the inversion should be recognized and this option is unnecessary.

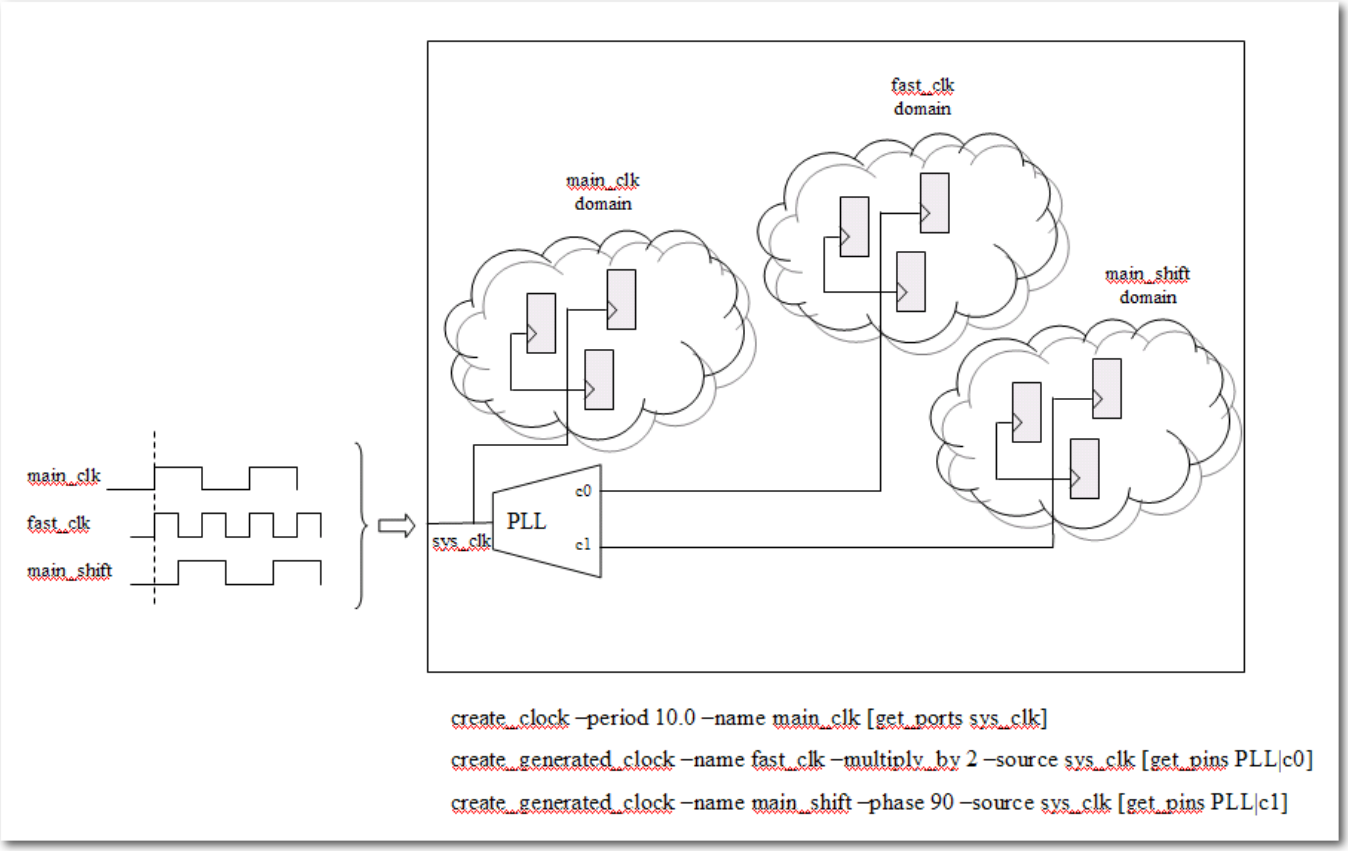
-source - This option specifies the physical point in a design where the generated clock's waveform is derived from. Note that the -source is not a clock but a physical name in the design. More often than not, a user will enter the <target> of the upstream master clock or generated clock, but this is not a requirement as the <target> can be any point between the s previous clock(*create_clock* or *create_generated_clock*). A good example of this is when *derive_pll_clocks* calls its *create_generated_clock* assignments, the -source option is the input pin of the PLL. This properly grabs the waveform at that point and the generated clock's waveform will be based on this, but the delay to that point will still properly start at the FPGA input. The beauty of this is that the assignment doesn't have to know the name of what drives the generated clock. The PLL could be driven by an input clock port with any name chosen by the designer, or by another PLL, or pretty much any clock source, and the assignment would still work.

-master_clock - If the specified -source option has more than one clock traveling along it, then -master_clock is required to specify which clock this generated clock is based on. For example, the output of a clock mux might have two clocks going through it. If there is a *create_generated_clock* assignment downstream from the mux, the user would use -master_clock to specify which of the two clocks this generated clock is based on.

-add - If the <target> already has a clock on it, the -add option is used to add this generated clock. Without it, TimeQuest will ignore the new constraint and issue a warning. This is generally used with clock muxes, where multiple clocks go through a single node. Another use is with the PLL's clock switchover(which is similar to a clock mux), where each output of the PLL can be driven by one of two input sources, and hence there are two generated clocks applied to each output and the -add option is used.

How Generated Clocks are Analyzed

Generated clocks are analyzed as if they were coming into the device where the upstream *create_clock* is applied. Take a look at the following diagram:



This design has a 10ns clock coming into the FPGA, which feeds a PLL where two generated clocks come out, one that is 2x the frequency and one that is phase-shifted 90 degrees. Now, most users think the PLL "does something" to the source clock, so the main_clk comes into the PLL and the fast_clk and main_shift clocks come out. This is not how it is timed. Instead, it will look like three clocks come into the FPGA. The main_clk feeds all the registers in main_clk domain, fast_clk feeds all the registers in fast_clk domain, and main_shift feeds all the registers in main_shift domain. Main_clk and fast_clk are edge-aligned, while main_shift has a 90 degree phase shift. As a result, all clocks modifications will be represented by the Launch and Latch edges during timing analysis. Look at Correlating Constraints to the Timing Report to see more of this. Although many users accept this, it confuses some since they expect to see something like a manual phase-shift show up in the PLL delays. In the example above, they might expect to see main_clk as the original waveform and then a shift occur as it goes through the PLL to create the clock main_shift. Instead, it looks like main_shift is coming into the FPGA and feeds the registers clocked by PLL|c1.

derive_pll_clocks

Note: In TimeQuest, type "derive_pll_clocks -long_help" for more information.

This command is not a true .sdc command, but calls out true .sdc commands. When generating a PLL, the user must enter how each output is to be generated. Because of this, TimeQuest knows how each PLL output should be constrained, and can therefore apply the proper *create_generated_clock* assignment to each PLL output.

The command does more than constrain PLL outputs; it also configures clocks used in the dedicated transceivers and adds multicycles between user logic and True LVDS SERDES. . Note that after it is read in, the TimeQuest messages can be expanded to show every command run by *derive_pll_clocks*, and as such, it is recommended every user run this command on their design to see what it does. Personally, I recommend having this in the main .sdc of every design.

I find many users don't want to use this constraint and prefer entering the individual constraints into their .sdc file. Technically, the two may be equivalent, but I have seen enough users modify their PLLs and forget to modify their .sdc that I strongly recommend sticking with *derive_pll_clocks.* There is a PLL-cross check warning if the user constrains a PLL in a manner different than how it was configured, but it is easy to miss warnings.

The one advantage to not running this command is that the user can name the clock whatever they want, rather than using the long, hierarchical PLL name chosen by *derive_pll_clocks*. That being said, I feel the advantages of *derive_pll_clocks* outweigh this.

If the user wants to put generated clock assignments on some of their PLL outputs and let *derive_pll_clocks* do the rest, they need to put the their *create_generated_clock* assignments before *derive_pll_clocks* to make sure they take priority. This is discussed in priority of derived assignments.

Options:

-create_ base_clocks - This will add a *create_clock* assignment to the clock ports driving the PLLs. If the system's clocks are all from input clocks that directly feed PLLs, then this single line can constrain all the clocks in the FPGA. I generally use individual *create_clock* assignments for clocks driving the PLLs, but it is up to the user.

-use_tan_name - This option is used when converting constraints from the Classic Timing Analyzer(TAN) to TimeQuest. TAN named it's PLL outputs in a format different than TimeQuest's *derive_pll_clock* command normally does. If the user had another assignment that was converted from Classic that referenced these clock names, they would no longer match. This option is used to prevent that from happening. Unless it's necessary, I recommend not using this, since new designs do not need this and it just adds confusion as other users ask why this

option was on. In reality, most designs from TAN didn't reference the PLL clock names, and this option wouldn't be necessary with those designs.

derive_clock_uncertainty

Note: In TimeQuest, type "derive_clock_uncertainty -long_help" for more information.

For all devices on 65nm and newer, this option should be in used in every project. It applies clock uncertainty between clock domains based on device characterization and models clock issues like PLL jitter, but is not limited to PLL clocks. Much like derive_pll_clocks, the command calls out individual *set_clock_uncertainty* for every clock transfer, and these assignments can be found in the TimeQuest messages.

Options:

This command calls out set_clock_uncertainty assignments between all clocks in the user's design, based on characterization. The options deal with prioritization if the user has their own *set_clock_uncertainty* assignments.

-add - Adds derived uncertainty to any uncertainties explicitly added by the user.

-overwrite - Overwrites the user's uncertainty, independent of order.

In most designs the user does not need to manually enter any uncertainty, and so a single call to *derive_clock_uncertainty* is all that is needed.

derive_clocks

Note: In TimeQuest, type "derive_clocks -long_help" for more information.

This is a shortcut command to quickly constrain incoming clocks without having to know what ports they come in on. Under most circumstances, do not use this command. The assignment *derive_clocks* applies a clock with *create_clock* to all unconstrained clocks in the device, except for PLL outputs. Since only a single -period can be given for this option, it is not very useful if more than one clock period is coming into the device.

This command is not recommended, and used only for benchmarking small pieces of logic, where they user wants to constrain it without thinking about creating a .sdc file. It could also be used for something simple like a CPLD or small FPGA that only has one clock, but writing a *create_clock* directly makes more sense.

set_clock_groups

Note: In TimeQuest, type "set_clock_groups -long_help" for more information.

By default, all clocks are related in TimeQuest, and so paths going between different clock domains will be analyzed with a setup and hold relationship, and the fitter will try to close timing on those paths. Yet most designs have paths between unrelated clock domains. The *set_clock_groups* command is an eloquent way to tell TimeQuest what clocks are not related, and thereby cut timing on those paths. It basically creates groups of clocks, and any paths whose launch and latch clocks are in different groups will not be analyzed. The syntax looks like so:

```
set_clock_groups -asynchronous \
    -group { \
        adc_clk \
        the_adc_pll|altpll_component|auto_generated|pll1|clk[0] \
        the_adc_pll|altpll_component|auto_generated|pll1|clk[1] \
        the_adc_pll|altpll_component|auto_generated|pll1|clk[2] \
        } \
    -group { \
        sys_clk \
        the_system_pll|altpll_component|auto_generated|pll1|clk[0] \
        the_system_pll|altpll_component|auto_generated|pll1|clk[1] \
        }
    -group { \
        the_system_pll|altpll_component|auto_generated|pll1|clk[2] \
        }
```

This looks complex at first glance, but the more I use it, the more I realize how elegant of a command it is. For example, if I were to use *set_false_path* assignments to cut timing between each clock domain and try to mimic that above statement, it would take 38 individual assignments, which would be difficult to understand. Instead, I can look at this command and quickly ascertain which clocks are related.

Notes:

- Each -group is a list of clocks that are related to each other
- There can be as many *-group {}* as the user wants. If they need fifty groups, that's fine. If entering the constraint through Edit -> Insert Constraint, it only has space for two groups, but this is only a limitation of that GUI. Feel free to add more.
- User's look at the command and think it is grouping clocks, but TimeQuest relates all clocks by default so in essence, they're already in one big group. This command is really cutting timing between clocks in different groups within a set_clock_groups command.
- Any clock not listed in the assignment keeps the default of being related to all clocks
- A clock can only be in one -group in a single *set_clock_groups* assignment
- A user can have multiple set_clock_groups assignments
- PLL clock names get long, and so this command is unreadable if all clocks are on a single line. Instead, make use of the Tcl escape character "\". By putting a space after your last character and then "\", the end-of-line character is escaped. (And be careful not to have any whitespace after the escape character, or else it will escape the whitespace, not the return character).

- For designs with complex clocking, writing this constraint can be an iterative process. For example, a design with two DDR3 cores and high-speed transceivers could easily have thirty or more clocks. In those cases, I just add the clocks I create and whose relationships I understand into the set_clock_groups command. Since clocks not in the command are still related to every clock, I am conservatively grouping what I know while leaving everything else related. If there are still failing paths in the design between unrelated clock domains, I start adding in the new clock domains as necessary. In this case, a large number of the clocks won't actually be in the set_clock_groups command, since they are either cut in the IP's .sdc file(like the ones generated by the DDR3 cores), or they only connect to clock domains they are related to.

- I generally leave virtual clocks created for I/O analysis out of this constraint. The only clocks they connect to are real paths, so there is no need to cut their analysis to other clocks.

- The option after *set_clock_groups* is either *-asynchronous* or *-exclusive*. The -asynchronous flag means the clocks are both toggling, but asynchronously to each other. The -exclusive flag means the clocks do not toggle concurrently, and hence are mutually exclusive. A good example of this might be a clock mux that has two generated clock assignments on it. Since only one can toggle at a time, these clocks are -exclusive. TimeQuest will analyze your design identically for either flag. This option is really used for ASICs, where SI issues like cross-talk between toggling clocks are analyzed. The -asynchronous option means cross-talk can occur, while the -exclusive option means it cannot. If going to Hardcopy, which uses ASIC analysis tools on the back-end, it is recommended to get this right. For FPGAs it does not matter since the analysis is the same. The more conservative value is *-asynchronous*, since this states the clocks can interfere with each other.

- If *set_clock_groups* has a single group, then the clocks in that group are implicitly cut from all other clocks in the design. For example, the user could have:

set_clock_groups -asynchronous -group {clk_a clk_b}
set_clock_groups -asynchronous -group {clk_c clk_d}

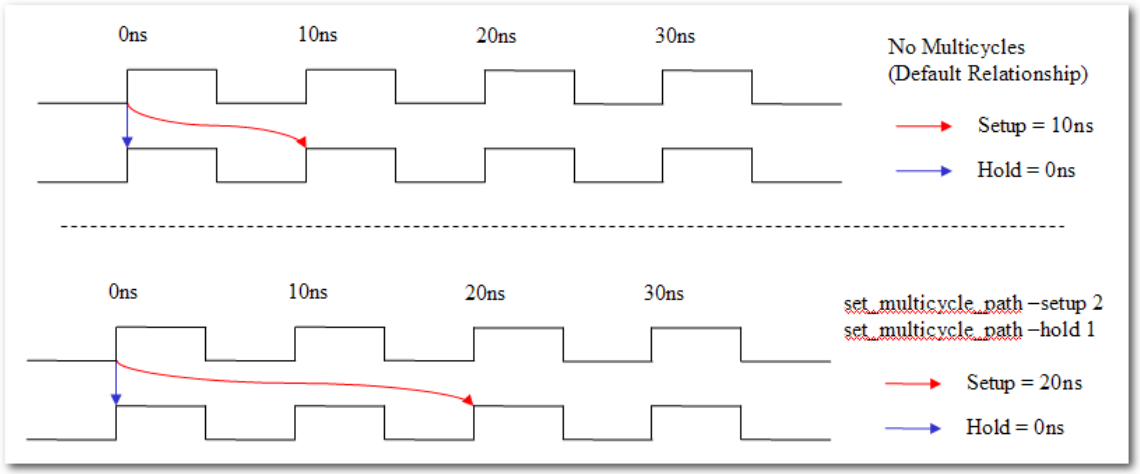
The first command cuts clk_a and clk_b from all other clocks in the design, but clk_a and clk_b are still related. The second command does the same for clk_c and clk_d. This special syntax is not recommended because the cuts are implicit without detailing the other clocks. This can lead to errors where users unintentionally cut timing between related clock domains.

- A quick tip for writing set_clock_groups can be found here.

set_multicycle_path

Note: In TimeQuest, type "set_multicycle_path -long_help" for more information.

By default, all clocks are related in TimeQuest and hence a default setup and hold relationship will be found. This default relationship is not always what the user wants, and multicycles allow the user to change this relationship. The key point of multicycles is that they are still based on the clock edges, and the user is just specifying different edges. For example, the following diagram shows the default relationship between two clocks, and the relationship after adding a multicycles setup of 2 and multicycles hold of 1:



Since the changes are based on the clock edges, if the user changes their input clock period from 10ns to 8ns, the multicycles requirement will change with it, and hence the second waveform would automatically have a setup relationship of 16ns instead of 20ns. I have seen designs with hundreds of multicycles, and with a single modification of their input clock period, all of their internal requirements are automatically updated.

How multicycles affect the default relationships are shown here. Be aware that there are two common cases of multicycles, and most users get by with just understanding these two cases.

Multicycles can be between keepers, i.e. between registers, I/O ports, etc. They can also be between clocks, in which case all transfers between those clock domains are affected by the multicycle. For the two common cases, opening the window is usually done between keepers, and reflects that the behavior on that logic is different than the default relationship. The second case, shifting the window, is usually done between clocks, since the default clock relationship is not the user's intent.

Options:

Most of the multicycle options control what paths the multicycles is applied to. For example:

```
# Opens the window from halfrate_src to halfrate_dst
set_multicycle_path -setup -from *halfrate_src* -to *halfrate_dst* 2
set_multicycle_path -hold -from *halfrate_src* -to *halfrate_dst* 1

# Open the window to data driving flash device:
set_multicycle_path -setup -to [get_ports Flash_Data*] 4
set_multicycle_path -hold -to [get_ports Flash_Data*] 3

# Shifts the window for all transfers between clock domains a and b:
set_multicycle_path -setup -from [get_clocks clk_a] -to [get_clocks clk_b] 2
```

The above examples show different types of filters. The first one is between registers in the design(and could have used get_registers or get_keepers, but I wanted to show an example

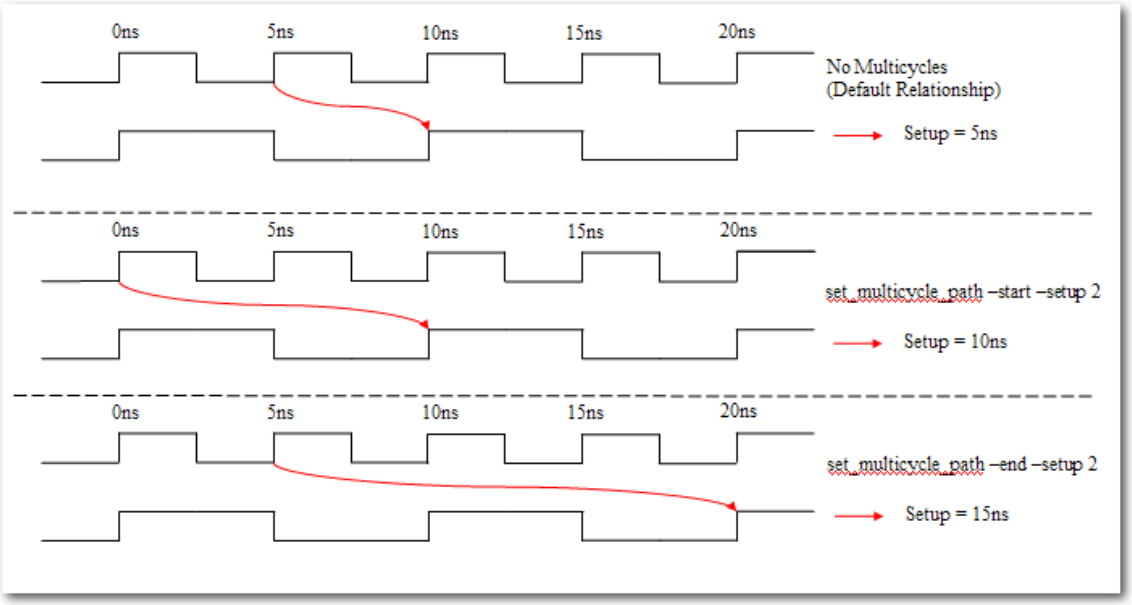
without it.) The second example is a multicycle on all paths to an I/O bus, and the third example is between clocks, so that every path between these two domains gets multicycled.

-from/-rise_from/-fall_from - These options control the source. -from is inclusive of all rising edge registers and falling edge registers, while -rise_from and -fall_from allow the user to only multicycle paths on registers that are clocked on the rising or falling edge. If no option is specified, then all sources are allowed, i.e. "*-from* *"

-to/-rise_to/-fall_to - These work in a similar manner, where -to is inclusive, getting registers clocked on both the falling and rising edge, and -rise_to/-fall_to filter to registers that are clocked on the rising or falling edge. If no option is specified, then all destinations are allowed, i.e. "*-to* *".

The <value> used for a multicycle refers to the edge count. The default setup relationship is called the "1" edge, and the default for hold relationship is called the "0" edge. As these values increase, the relationship gets looser, i.e. the setup relationship gets more positive and the hold relationship gets more negative. This is all covered in more detail in the section on determining multicycle relationships.

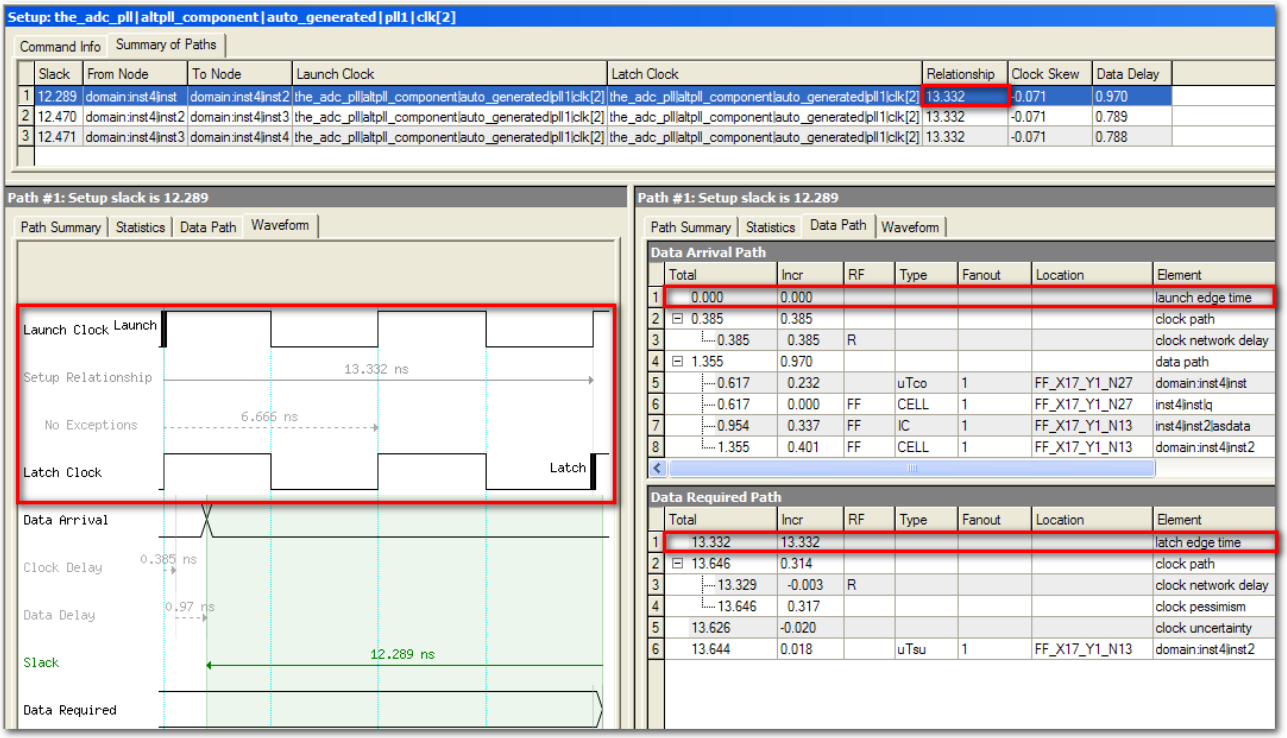
-start/-end - This option determines which clock's period, the launch or latch, is used to modify the default relationship. If no option is specified, the default is -end. The following diagram shows the default setup relationship from a 5ns clock to 10ns clock. (The line is drawn from 5ns to 10ns, but the setup relationship is the difference between these two values).



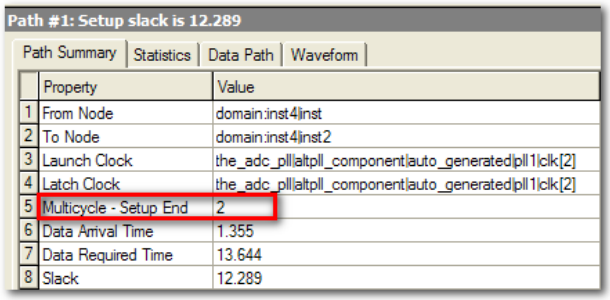
The middle waveform shows what happens when a multicycles -setup 2 is applied using -start. The "start" of the arrow is moved back one clock cycle, increasing the setup relationship by the period of the launch clock. The third waveform shows what happens when -end is used. The end of the arrow is moved forward one clock period, increasing the setup relationship by one period of the destination clock. The user should use whatever is appropriate to reflect how their design works.

This option only matters if the period of the source clock is different than the destination clock. If they're the same, the user gets the same result using -start or -end.

One last point is to show how multicycles affect the timing reports of *report_timing*. The next example has clocks with a period of 6.666ns, but there is a multicycles -setup 2 applied to the paths:



Red rectangles show everything that has changed due to this multicycles. The Summary at the top now has a relationship of 13.332ns instead of 6.666ns The Data Path tab on the bottom right has a launch edge time of 0ns and a latch edge time of 13.332ns. The Waveform tab on the bottom left shows that the Latch edge is now the second rising edge, resulting in a Setup Relationship of 13.332ns. The Waveform even shows the 6.666ns relationship if there were No Exceptions, i.e. no multicycles. Finally, the Path Summary tab clearly states a Multicycle was applied:



get_fanouts

set_max_delay/set_min_delay

Note: In TimeQuest, type "set_max_delay -long_help" or "set_min_delay -long_help" for more information.

These two constraints act as low-level overrides of the setup and hold relationships. The constraint set_max_delay overrides the setup, while set_min_delay overrides the hold relationship. Note that these constraints are not point-to-point requirements between registers, which is a common misperception, and clock skew is still used in calculating slack. These constraints are similar to multicycles, but rather than being based on edges of the existing clock, they are based solely on the <value> entered by the user. If a user applies a set_max_delay of 8ns between two registers, the user can modify their source and/or destination clock properties in their SDC file, and it will have no affect on the slack calculation for that path.

Options:

-from/-rise_from/-fall_from - These options control the source. -from is inclusive of all rising edge registers and falling edge registers, while -rise_from and -fall_from allow the user to only multicycle from registers clocked by the rising or falling edge. If no option is specified, then all sources are allowed, i.e. "*-from* *"

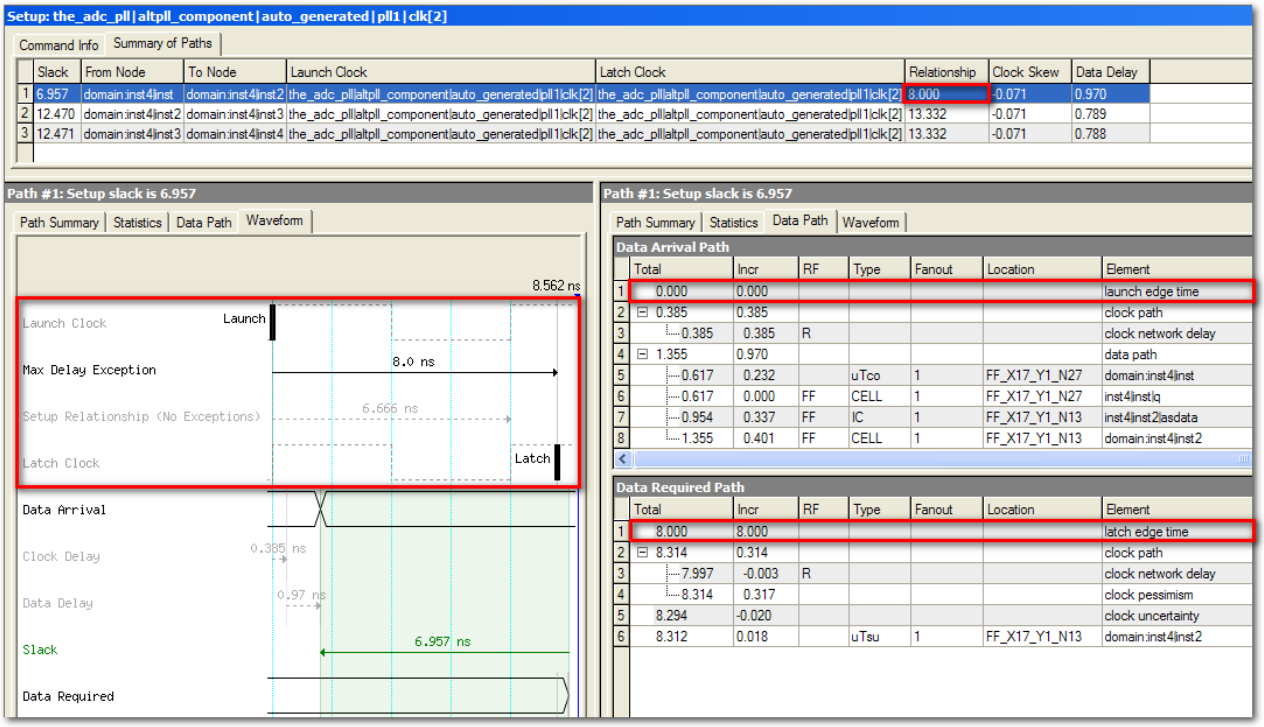
-to/-rise_to/-fall_to - These work in a similar manner, where -to alone is inclusive, getting registers clocked on both the falling and rising edge, while -rise_to/-fall_to filter the multicycle to apply to destinations clocked by the rising or falling edge. If no option is specified, then all destinations are allowed, i.e. "*-to* *".

<value> - This is the override value, in nanoseconds.

These values show up in *report_timing* in the same manner as multicycles. The following example uses the same paths from the previous multicycle example but applied:

set_max_delay 8.0 -from domain:inst4|inst -to domain:inst4|inst2

Running *report_timing -setup* shows the following:



The setup Relationship is now 8.0ns, where on the previous example it was 13.332ns. The launch edge time becomes 0ns, and the latch edge time becomes the <value> entered of 8ns. The Waveform view on the bottom left is a good visualization, in that the launch and latch edge times are now independent of the clock waveform.

The *set_max_delay* and *set_min_delay* constraints have two dangers that users should be aware of, described here. These constraints can also be used for device-centric I/O constraints, specifically Tsu, Th, Tco and Tpd constraints, which are described here.

set_false_path

Note: In TimeQuest, type "set_false_path -long_help" for more information.

This command tells TimeQuest not to analyze a path or group of paths. It can be between keepers(registers, I/Os, etc.) or between clocks. When the constraint is applied to clocks, then all paths that are clocked by the respective clock will not be analyzed. Three examples:

 # Cut timing from an input port to all of its destinations:
set_false_path -from [get_ports reset_button]

 # Cut timing from a mode_select register, which is static in the design, to all of its destinations:
*set_false_path -from [get_keepers *|mode_select]*

 # Cut timing from clk_a to clk_b:
set_false_path -from [get_clocks clk_a] -to [get_clocks clk_b]

The last example cuts timing on all paths where clock clk_a drives the source register and clock clk_b drives the destination register. Note that transfers in the other direction have not been cut, and another *set_false_path* assignment would be necessary. Cutting timing between clocks is often best accomplished with *set_clock_groups*.

set_clock_uncertainty

Note: In TimeQuest, type "set_clock_uncertainty -long_help" for more information.

When clocks are created, they are ideal and have perfect edges. This constraint is used to add uncertainty to those perfect edges, and mimic clock-level effects like jitter. In general, most designs never use this constraint and rely on *derive_clock_uncertainty*, which models all internal clock effects for the user. If a user does want to use this, I suggest they use the -add option, so their uncertainty is additive to that calculated by *derive_clock_uncertainty*.

set_clock_uncertainy applied to a clock does not have its uncertainty propagate to generated clocks downstream. The user needs to apply uncertainty to those clocks too, if that is how they want it analyzed.

set_clock_latency

Note: In TimeQuest, run "set_clock_latency -long_help" for more information.

This is a cool command that models board-level clock delays, although admittedly, I seldom see it used. The basic syntax looks like so:

set_clock_latency -source -late 1.234 sys_clk
set_clock_latency -source -early 1.1 sys_clk

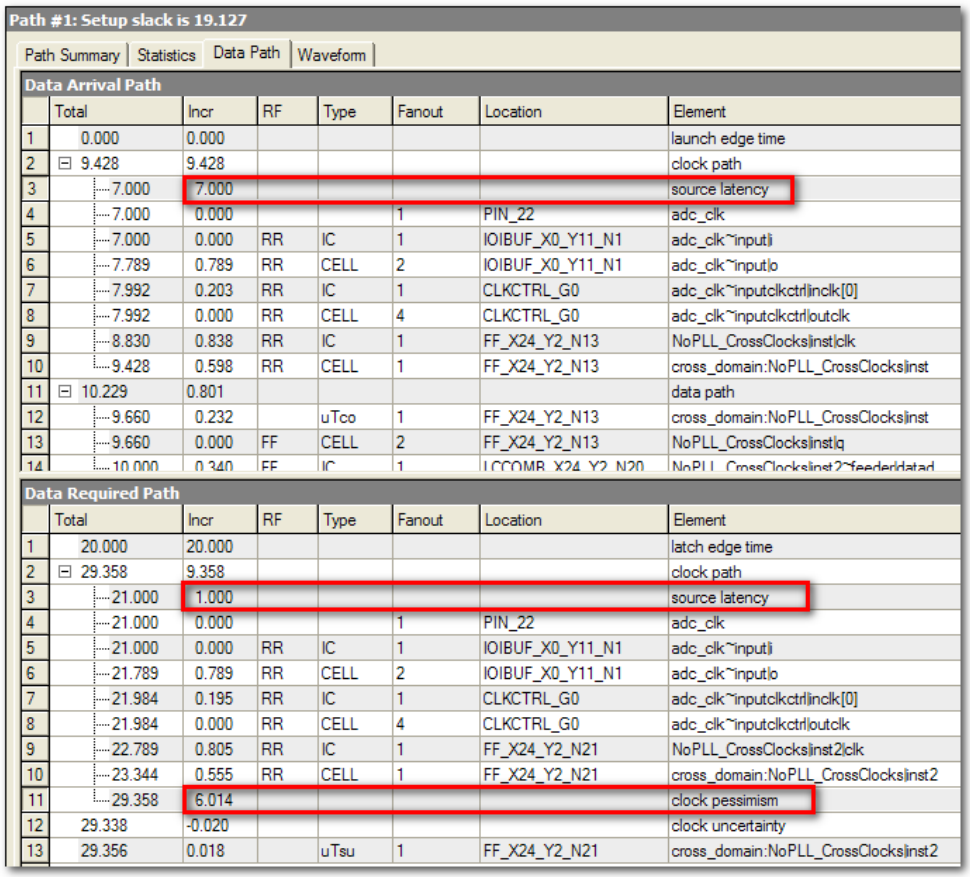
With such constraints applied to a clock, TimeQuest knows the board-level clock delay to sys_clk can be as late as 1.234ns and as early as 1.1ns. Where this is most useful is for I/O constraints, where the user can specify the clock latency to the FPGA clock port, as well as the clock latency to the virtual clock. TimeQuest will use the correct value when doing setup and hold slack analysis. For example, on an output port, it will use the late clock latency to the FPGA's clock and early clock latency to the external virtual clock. This analyzes the worst case scenario where the data arrival path is as long as possible, and the data required path is as short as possible. Likewise, for hold checks it will use the early value for the clock to the FPGA, and the late value for the delay to the external virtual clock.

The second use for set_clock_latency is on feedback clocks, where a clock goes out an FPGA port and then comes back. This scenario would have a generated clock on the output port with -source from the clock driving it, and another generated clock on the input port, whose -source would be the output port. This input generated clock needs a *set_clock_latency* assignment to show the external delays from the output to the input.

One slightly annoying thing with set_clock_latency is that the -early and -late values are used for internal clocks, and then removed through common clock path pessimism. For example, let's say a design had the following:

set_clock_latency -source -early 1.0 adc_clk
set_clock_latency -source -late 7.0 adc_clk

Now, -early and -fall times would never really vary by that much, but I made them large for illustration purposes. Let's run *report_timing -setup* on a path inside the FPGA clocked by adc_clk. Highlighted below, the source register gets the 7ns late latency and the destination register gets the 1ns early latency. This is not what we want, as the clock does not really vary by 6ns cycle to cycle within the FPGA. Note later on, the clock pessimisim adds 6.014ns back to the path. This completely accounts for the 6ns difference. The extra 14ps is for pessimism inside the FPGA, and would have been there regardless of using *set_clock_latency*. In the end, the *set_clock_latency* had no affect within the clock domain, and only has an affect when relating to other clocks with different latencies, which is how board-level clock latencies should work. The problem is that the math looks confusing in that it uses different latencies and then backs them out with clock pessimism.



Path #1: Setup slack is 19.127

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 9.428	9.428					clock path
3	7.000	7.000					source latency
4	7.000	0.000			1	PIN_22	adc_clk
5	7.000	0.000	RR	IC	1	IOIBUF_X0_Y11_N1	adc_clk~input|i
6	7.789	0.789	RR	CELL	2	IOIBUF_X0_Y11_N1	adc_clk~input|o
7	7.992	0.203	RR	IC	1	CLKCTRL_G0	adc_clk~inputclkctrl|inclk[0]
8	7.992	0.000	RR	CELL	4	CLKCTRL_G0	adc_clk~inputclkctrl|outclk
9	8.830	0.838	RR	IC	1	FF_X24_Y2_N13	NoPLL_CrossClocks|inst|clk
10	9.428	0.598	RR	CELL	1	FF_X24_Y2_N13	cross_domain:NoPLL_CrossClocks|inst
11	⊟ 10.229	0.801					data path
12	9.660	0.232		uTco	1	FF_X24_Y2_N13	cross_domain:NoPLL_CrossClocks|inst
13	9.660	0.000	FF	CELL	2	FF_X24_Y2_N13	NoPLL_CrossClocks|inst|q
14	10.000	0.340	FF	IC	1	LCCOMB_X24_Y2_N20	NoPLL_CrossClocks|inst2~feeder|datad

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	20.000	20.000					latch edge time
2	⊟ 29.358	9.358					clock path
3	21.000	1.000					source latency
4	21.000	0.000			1	PIN_22	adc_clk
5	21.000	0.000	RR	IC	1	IOIBUF_X0_Y11_N1	adc_clk~input|i
6	21.789	0.789	RR	CELL	2	IOIBUF_X0_Y11_N1	adc_clk~input|o
7	21.984	0.195	RR	IC	1	CLKCTRL_G0	adc_clk~inputclkctrl|inclk[0]
8	21.984	0.000	RR	CELL	4	CLKCTRL_G0	adc_clk~inputclkctrl|outclk
9	22.789	0.805	RR	IC	1	FF_X24_Y2_N21	NoPLL_CrossClocks|inst2|clk
10	23.344	0.555	RR	CELL	1	FF_X24_Y2_N21	cross_domain:NoPLL_CrossClocks|inst2
11	29.358	6.014					clock pessimism
12	29.338	-0.020					clock uncertainty
13	29.356	0.018		uTsu	1	FF_X24_Y2_N21	cross_domain:NoPLL_CrossClocks|inst2

set_input_delay/set_output_delay

Note: In TimeQuest, type "set_input_delay -long_help" or "set_output_delay -long_help" for more information.

These are the two dedicated commands for constraining I/O. Personally, I find it better to not think of them as constraints at all. Instead they describe a circuit outside of the FPGA and that circuit, coupled with the circuit inside the FPGA, creates a full setup and hold analysis. The steps for creating these constraints are found in the Getting Started - I/O Timing section. Please look at that section before using this constraint.

Options:

<target> - This is the port the constraint is applied to. This means there is a register external to the FPGA connected to this port.

-clock - This is the clock driving this external register. In almost all cases this clock should be a virtual clock. The only major exception is for source-synchronous outputs, where the -clock should be the name of a *create_generated_clock* that is applied to the port driving out the clock.

-max/-min - This is the external delay to this external register. The -max value affects the setup analysis, and the -min value affects the hold analysis. As the -max value increases, the setup requirement gets tighter because the FPGA's internal delays must get smaller in order to meet the setup relationship between clocks. Likewise, as the -min value decreases, the hold requirement gets tighter, because the FPGA must add more delay in order to meet the hold relationship between clocks. This makes sense, as the difference between -max and -min grows, then a larger percentage of the data period is being used externally, and the FPGA's delays must tighten.

-reference_pin - This option is for *set_output_delay* only, and meant to reference the output port that a clock goes out on, mainly for source-synchronous outputs. After doing many of these interfaces, I recommend not using this option at all, and instead always putting a *create_generated_clock* assignment on the output port driving out the clock, and referencing that clock with the *set_output_delay -clock* option. By putting a generated clock on the output and referencing that, the user can achieve identical analysis to the *-reference_pin* option, but can do a lot more if need be. The ability to constrain source-synchronous outputs in two different way probably adds more confusion than helps, and so I just recommend against using *-reference_pin.* (Although if it is in your design and working, there is nothing explicitly wrong with this option).

-clock_fall - This option states that the external register is clocked on the falling edge of the clock. This naturally affects the setup and hold relationships to clocks inside the FPGA. This option is most commonly used on double-data rate interfaces, where -add_delay is also used.

-add_delay - This option does not mean to add the delay from this constraint to any previous external delays. In reality it means there is another external register connected to the port. This is most commonly used with double-data rate interfaces, and often looks like so:

```
set_input_delay -clock ext_clk -max 0.5 [get_ports {ddr_data[*]}]
set_input_delay -clock ext_clk -min -0.5 [get_ports {ddr_data[*]}]
set_input_delay -clock ext_clk -max 0.5 [get_ports {ddr_data[*]}] -clock_fall -add_delay
set_input_delay -clock ext_clk -min -0.5 [get_ports {ddr_data[*]}]-clock_fall -add_delay
```

The values of 0.5 and -0.5 were chosen arbitrarily. The above constraints basically state that each input port of bus ddr_data is driven by two external registers, one clocked by the rising edge of ext_clk(this is done in the first two lines) and one clocked by the falling edge(the last

two lines). Without -add_delay on the last two lines, they would override the first two lines and a warning would be issued.

set_max_skew

Note: In TimeQuest, type "set_max_skew -long_help" for more information.

This is not a true SDC constraints, and was added to TimeQuest because it was commonly requested. It must be used in conjunction with *report_max_skew*. This command constrains the skew, and running *report_max_skew* in TimeQuest will give a report of everything that has been constrained. (Quartus II 10.0 added a Task called Report Max Skew Summary that can easily be clicked on, rather than manually typing *report_max_skew*, but it only gives a summary. Re-run the command with -detail set to full_path to get a detailed analysis*).* There is also a reporting command called *report_skew*, which reports the skew on specified paths, but won't actually constrain them during a compile. That command is useful for experimenting with skew analysis, and when the user feels they have it right, using its parameters with *set_max_skew*.

Note that I have often been asked how to constrain the skew of something, and when I ask more, realize the user really wants regular setup and hold analysis. Source synchronous interfaces are the common scenario, whereby the user claims they want to constrain the outputs to have a specific skew, when in reality they want to constrain their data in relation to the clock going off chip. My point is that user's should look at the original SDC constraints before using *set_max_skew*.

Pretty much every real use I've seen involved inputs or outputs. On inputs, it's usually a signal feeding multiple registers, all clocked by different phases of the same clock. In essence it's an oversampling circuit, or a timing circuit(an edge comes in and the user is trying to time it as accurately as possible). The command looks like:

set_max_skew -from [get_ports din] 0.5

Then in the analysis the user would run something like:

report_max_skew -npaths 20 -detail full_path -panel_name "Skew"

The second common case is when the user has multiple outputs they want aligned, and are not feeding a synchronous interface(if they are feeding anything synchronous, skew is not the correct command to use). The constraint might look like:

set_max_skew -to [get_ports led_data] 0.5*

Notes:

- The skew command is not just the datapath, but also includes the clock driving the launch and/or latch registers. This can be modified with the -include/-exclude options.
- There are -include and -exclude options that give the user much control over what is calculated for skew. Part of this is because there is not a consensus on how to report skew. For

example, if the user is controlling skew on a single input being clocked by multiple registers at different phases of the same clock. Should the capture register's micro-parameters of µTsu and µTh be included against the slack budget, whereby it would add in µTsu for the long path and subtract µTh for the short path, making the skew longer? Now technically these micro-parameters aren't a difference in the data delay, but on the other hand if the inputs is asynchronous it will transition at times that violate these register's µTsu/µTh, causing it to go metastable. Different scenarios deal with this in different ways, so it has been left up to the user. (And again, if the input isn't really asynchronous, the user should not be using the skew constraint).

- This is discussed in the fitter section, but note that the placer will not optimize for skew and will try to place all signals as close together as possible. This can lead to a non-balanced placement whereby two destinations of an input port might be placed in the LAB next to the register and two other destination registers are placed a LAB over. (The four registers can't be placed in the same LAB because only two clocks can feed a LAB, and each register has its own clock) It is the router that then adds delays and try to meet the skew requirements, but if the placement is non-balanced, routing delays may be too coarse and the results may not be good. I generally suggest hand-placing the registers with LAB location assignments, getting a balanced placement that the router can then work with. This is relatively easy to do and gets better results than relying completely on the fitter.

- The report can be a little confusing at first. Below is a screen shot of the skew on an input feeding multiple registers:

skew

Command Info | Summary of Paths

	Name	Slack	Required Skew	Actual Skew	From Node	To Node	Launch Clock	Latch Clock
1	set_max_skew	-0.423	0.500	0.923	[get_ports {in_skew}]			
2	Skew for the Latest Arrival							
3	--	-0.423	0.500	0.923	in_skew	inst9	n/a	the_system_pll|altpll_component|auto_generated|pll1|clk[1]
4	--	-0.348	0.500	0.848	in_skew	inst17	n/a	the_system_pll|altpll_component|auto_generated|pll1|clk[2]
5	--	-0.177	0.500	0.677	in_skew	inst14	n/a	the_system_pll|altpll_component|auto_generated|pll1|clk[0]
6	Skew for the Earliest Arrival							
7	--	-0.423	0.500	0.923	in_skew	inst14	n/a	the_system_pll|altpll_component|auto_generated|pll1|clk[0]
8	--	-0.219	0.500	0.719	in_skew	inst9	n/a	the_system_pll|altpll_component|auto_generated|pll1|clk[1]

Path #1: Slack is -0.423 (VIOLATED)

Path Summary | Path Statistics | Data Path

Latest Path Arrival

	Total	Incr	RF	Type	Fanout	Location	Element
1	Data Arrival						
2	0.000	0.000	R				clock network delay
3	0.000	0.000	F		1	PIN_34	in_skew
4	0.000	0.000	FF	IC	1	IOIBUF_X0_Y5_N15	in_skew~input|i
5	0.934	0.934	FF	CELL	3	IOIBUF_X0_Y5_N15	in_skew~input|o
6	1.643	0.709	FF	IC	1	FF_X0_Y5_N17	inst9|d
7	1.883	0.240	FF	CELL	1	FF_X0_Y5_N17	inst9
8	Clock Arrival						
9	2.073	0.190	R				clock network delay
10	2.178	0.105		uTsu	1	FF_X0_Y5_N17	inst9

Earliest Path Arrival

	Total	Incr	RF	Type	Fanout	Location	Element
1	Data Arrival						
2	0.000	0.000	R				clock network delay
3	0.000	0.000	R		1	PIN_34	in_skew
4	0.000	0.000	RR	IC	1	IOIBUF_X0_Y5_N15	in_skew~input|i
5	0.829	0.829	RR	CELL	3	IOIBUF_X0_Y5_N15	in_skew~input|o
6	1.414	0.585	RR	IC	1	LCCOMB_X1_Y5_N0	inst14~feeder|datab
7	1.742	0.328	RR	CELL	1	LCCOMB_X1_Y5_N0	inst14~feeder|combout
8	1.742	0.000	RR	IC	1	FF_X1_Y5_N1	inst14|d
9	1.811	0.069	RR	CELL	1	FF_X1_Y5_N1	inst14
10	Clock Arrival						
11	1.441	-0.370	R				clock network delay
12	1.255	-0.186		uTh	1	FF_X1_Y5_N1	inst14

The Latest Arrival Path below shows the Data Arrival minus the Clock Arrival of the longest path to a register, resulting in 2.178ns. The Earliest Arrival Path is the also Data Arrival minus Clock Arrival of this early path, resulting in 1.255ns. The difference is 0.923ns, or the Actual Skew shown at the top.

- Calculated skews tend to be larger than most users expect. This is because the skew calculation includes On-Die Variation(ODV). Without ODV, which was not in device models before the 65nm node, skew values looked extremely small. Without ODV, I have seen skew values reported to be less than 10ps. This is unrealistic in hardware. Likewise, I have seen Altera FPGA's skew compared to other FPGA's that do not model ODV, which makes Altera's look bad, but ODV is a significant component of skew and must be taken into account for realistic analysis. This is discussed in detail in the On-Die Variation section.

Constraint Priority

There are three ways that constraints co-exist, and it's probably best to understand them in that context. They are:

- Different constraints. What if a multicycles and a false_path are applied to a constraint? Which one has priority?

- Same constraint, different value. What if a path gets multicycles -setup assignments from two different places?

- Multiple assignments applied to a node

Priority between Different Constraints

The basic hierarchy of different constraints from lowest priority to highest priority:

1) *create_clock* and *create_generated_clock*

These constraints create clocks that drive registers, and as a result have a default setup and hold relationship.

2) *set_multicycle_path*

This constraint tells TimeQuest that the default setup and hold relationship is incorrect, and the user wants a different relationship based on the clock edges.

3) *set_max_delay* and *set_min_delay*

These constraints tell TimeQuest that the setup and hold relationships, whether determined by default or with multicycles, is incorrect, and the user wants the relationship to be an explicit value. Note that *set_max_delay* overrides the setup relationship and *set_min_delay* overrides the hold relationship, which are two mutually exclusive analyses.

4) *set_false_path* and *set_clock_groups*

These constraints tell TimeQuest not to analyze specific paths or clock transfers. Once a path has been cut by either of these commands, there is no way to un-cut it, i.e. these constraints have the highest priority.

You may note that I did not discuss *set_input_delay* and *set_output_delay*. This is because they are not really constraints in the classical sense, and instead describe a circuit outside of the FPGA. As such, they work in conjunction with these other constraints. For example, let's say I have a signal coming into the FPGA on port din, which goes through some combinatorial logic and out through dout. To constrain it, I might do something like:

create_clock -period 20.0 -name ext_clk
set_input_delay -clock ext_clk -max 4.0 [get_ports din]
set_output_delay -clock ext_clk -max 7.0 [get_ports dout]

(Note that I did not do -min delays. I am going to ignore hold time analysis for this example, but normally a design should have this too.) Anyway, the *set_input_delay* and *set_output_delay* describe registers outside of the FPGA and states they are clocked by ext_clk. As such, there is a default setup relationship of 20ns when this clock is the source and destination. This is the lowest priority. Since 11ns of delay are used externally, the FPGA must get its signal from din to dout in 9ns.

A user could then add a multicycles if that is too tight of a requirement:

s*et_multicycle_path -setup 2 -from [get_ports din] -to [get_ports dout]*

This multicycles has priority over the default clock relationship, and makes the setup relationship two clock periods, or 40ns. Since 11ns are used externally, the FPGA must get its data from din to dout in 29ns. If the .sdc also had:

set_max_delay -from [get_ports din] -to [get_ports dout] 30.0

This *set_max_delay* would have priority over the default setup relationship and the multicycled relationship, making the new setup relationship 30ns. Since 11ns are used externally, the FPGA must get its signal from din to dout in 19ns. Finally, if the .sdc had:

set_false_path -from [get_ports din] -to [get_ports dout]

The path is now no longer analyzed. This priority occurs independent of the order these commands are read in. As you can see, the *set_input_delay* and *set_output_delay* commands really just complete the circuit, and hence work with all the other commands.

One final note is the special case when *set_max_delay* and *set_min_delay* are applied to an I/O port that has no *set_input_delay* or *set_output_delay* assignment. As discussed, this special case will implicitly add a *set_input_delay* or *set_output_delay* constraint to the I/O with 0ns external delay and a clock called "n/a" behind the scenes. This only occurs if the user does not have a *set_input_delay* or *set_output_delay* constraint anywhere in their .sdc files. If they do, those constraints take priority over these implicit constraints.

Priority between Equal Constraints

This is when a path has two different multicycles assignments applied to it, or two different set_max_delay assignments. These could be from two different .sdc files, or two different levels of assignments. By levels, I mean a user might have the following two assignments in their .sdc files:

```
set_multicycle_path -setup -from top|domain1:inst_a|reg_a \
        -to top|domain1:inst_b|reg_b 4
set_multicycle_path -setup -from clk_a -to clk_b 2
```

If reg_a is clocked by clk_a, and reg_b is clocked by clk_b, then this would have two different assignments, one directly on the path and one between the clocks. The priority is quite simple, it is whatever constraint is read in last. Of course, knowing that may not always be so straightforward, and so it is recommended to run the task Report Exceptions. This command goes through the user's exception in their .sdc file and writes a report on if they were completely followed, partially, or incomplete.

Priority between Multiple Assignments to the Same Node

This is different than exception priority, in that an actual attribute is assigned to the node. These assignments include:

create_clock
create_generated_clock

set_input_delay
set_output_delay

All of these are applied to a node, and have different behavior. If a *create_clock* or *create_generated_clock* apply a clock to a node that already has a clock on it from a previous call of these commands, then the second clock will not be added. If the user wants multiple clocks on that node, then they should use the -add option for the second constraint.

Set_input_delay and *set_output_delay* assignments will overwrite any previous input or output constraints. If the user wants multiple delay constraints on the port, then they should use the -add_delay option for the latter assignments. Note that the -max and -min options are mutually exclusive and don't require -add_delay. A port feeding external DDR registers might have:

set_output_delay -clock ext_clk -max 0.5 [get_ports {ddr_data[*]}]
set_output_delay -clock ext_clk -min -0.5 [get_ports {ddr_data[*]}]
set_output_delay -clock ext_clk -max 0.5 [get_ports {ddr_data[*]}] -clock_fall -add_delay
set_output_delay -clock ext_clk -min -0.5 [get_ports {ddr_data[*]}]-clock_fall -add_delay

The first two lines are -max and -min, and since they are mutually exclusive(one affects setup analysis, the other is for hold analysis), there is no need for the -add_delay. The last two constraints would override the first two if not for the -add_delay option.

Finally, for both clock and I/O constraint conflicts, TimeQuest will issue a warning if the user has multiple assignments that are not resolved with the -add or -add_delay option. The warning for a second clock looks like so:

Warning: Ignored create_clock: Incorrect assignment for clock. Source node: adc_clk already has a clock(s) assigned to it. Use the -add option to assign multiple clocks to this node. Clock was not created or updated.

The warning for a second input/output delay constraint without -add_delay looks like so:

Warning: Assignment set_input_delay is accepted, but has the following problems:
Set_input_delay/set_output_delay has replaced one or more delays on port "adc_din_100".
Please use -add_delay option.

Priority between Derived Assignments and User Assignments

One common concern involves *derive_pll_clocks* and *derive_clock_uncertainty*, which are making *create_generated_clock* assignments and *set_clock_uncertainty* assignments for the user. These are handled in different ways due to what they are doing. The command *derive_pll_clocks* runs when it is immediately met, executing *create_generated_clock* assignments for each of the PLL output as if the user had them directly in their .sdc. If any of the PLL outputs already had a generated clock assigned to them earlier in the .sdc files, the command will not add a new assignment. If any generated clocks are applied to the PLL outputs

after *derive_pll_clocks* is called, the latter assignment is ignored with a warning, unless it has the -add option.

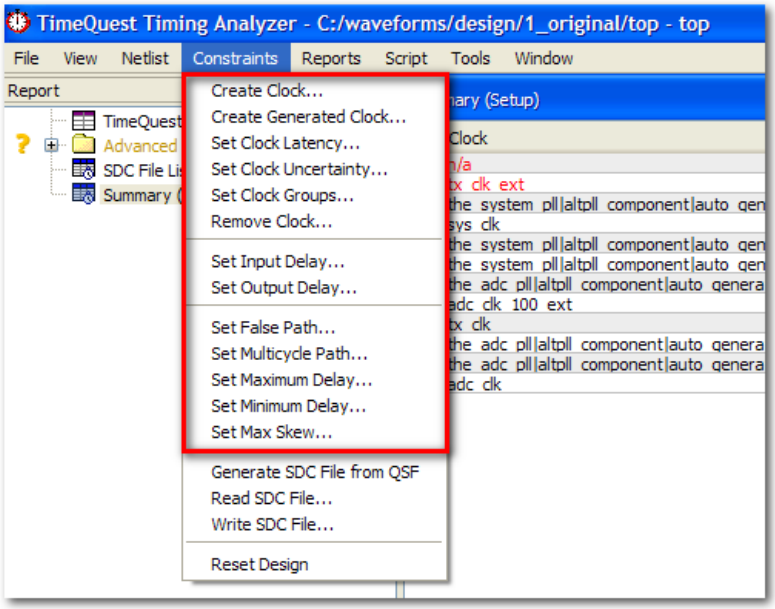
On the other hand, *derive_clock_uncertainty*'s individual calls of *set_clock_uncertainty* occur when the timing netlist is being updated, which is after all SDC files have been read in. If the user has *set_clock_uncertainty* assignments elsewhere in their .sdc files, those assignments will have priority. If the user's *set_clock_uncertainty* assignments or the *derive_clock_uncertainty* assignment has the -add option, then the uncertainties will be additive.

Section 4: The TimeQuest GUI

This section is not meant to give details on every option in the TimeQuest GUI, but instead is meant to get the user familiar and comfortable with analyzing their design. Because of that, the organization and recommendations will be based on how I use it. There is certainly room for disagreement on many of these topics, but I want to give an opinion that new users can evaluate. Starting off with an opinion…

Entering SDC Constraints from the GUI

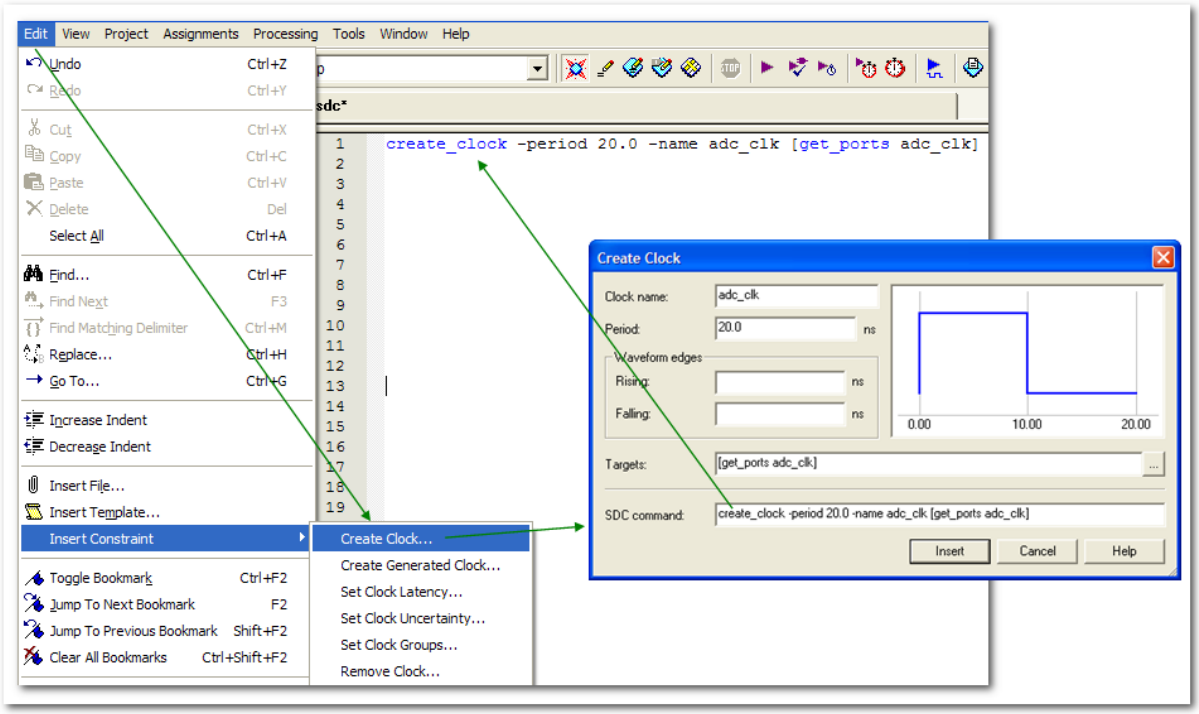
There are two ways to use the TimeQuest GUI for entering SDC constraints. Method #1 is directly from the main window's Constraints pull-down:



I recommend that users do not do this. It does not save the actual SDC constraint to a text file, so the user must go through extra steps to get the command into their .sdc file. Users end up using the Write SDC command, which makes their .sdc machine-generated, and prevents users from getting the many benefits of a user-created .sdc file, including useful comments, variables, constraint ordering, wildcards, etc. This method also applies the constraint directly to the database, which can cause difficulties in debugging priority issues. For example, a constraint run this way may work because it was applied after a clock was created, but when the user adds the constraint to their .sdc file it stops working, because they added it before the clock gets created.

The bottom line is this works as a quick method for testing a constraint by an expert user who is fully aware of what is occurring. Too many beginner/intermediate users run into trouble with these pull-down menus and so I recommend avoiding them altogether. Most importantly, there is another way to access these constraint dialogue boxes that is much, much better.

Open your .sdc file in Quartus II or TimeQuest Place the cursor where you want your constraint to be and go to Quartus II's pull-down menu Edit -> Insert Constraint. The exact same dialogue boxes are accessible and the constraint will be added, in plain text, to your .sdc file. It is not executed until the user reads the .sdc back into TimeQuest. This method is much easier to understand and really what the user should be doing. Method #2 is entering constraints into the .sdc file from the Quartus II/TimeQuest editor's Edit Insert Constraint:



Two quick notes on using the GUI for entering constraints:

- The GUI menus do not show every option available for a constraint, only the most commonly used ones. To see all options, type "*command –help*" in the TimeQuest GUI, e.g. "*create_clock –help*". Note that a user can point to a constraint in the .sdc editor and a tooltop will pop-up showing all the options.
- The command dialogue boxes are good for new users, but I find most users quickly abandoning them and cutting-and-pasting commands directly in their .sdc. The one great benefit of the dialogue boxes is the […] box that opens the Name Finder. This lets the user enter a wildcard and see if it matches anything in the design database, avoiding name-matching mistakes. This can be done from the main TimeQuest GUI's View -> Name Finder, which copies the name matching command to the user's clip-board, allowing the user to paste it into their .sdc.

Since this is about the options under TimeQuest's Constraints pull-down menu, I will briefly touch on the options that are not constraints:

Generate SDC File from QSF - This command is for designers that have constrains from the Classic Timing Analyzer(TAN) stored in their .qsf, and want to convert them to an SDC file so they can use TimeQuest. Note that this conversion is only a getting started point, and is not guaranteed to be 100% correct. The Quartus II Handbook has a whole section on converting

to TimeQuest. I have seen user's who understand the Classic Timing Analyzer try to convert their designs without learning TimeQuest, and have a very difficult time. I recommend learning TimeQuest up front, especially the Getting Started section of this guide. If you understand what TimeQuest is doing, it is much easier to convert a design. The other big issue I've seen is that the Classic Timing Analyzer allowed designers to constrain a design without understanding much about timing analysis. It basically makes a lot of assumptions, which are right in most cases, but problematic when they are wrong. Since users of the Classic Timing Analyzer often don't understand these assumptions, they don't really understand their original constraints and hence don't understand how to convert them. Again, learning TimeQuest and taking a more rigorous approach to static timing analysis is the recommended course of action.

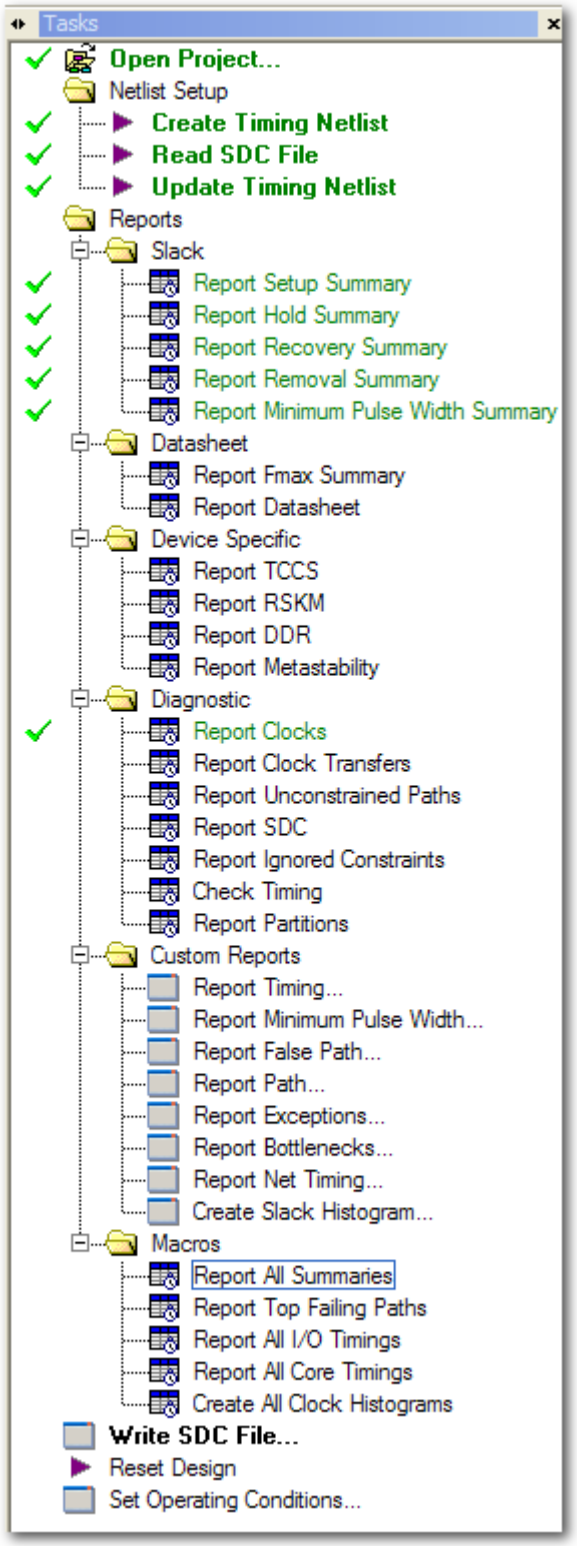
Read SDC File… - This allows the user to select an .sdc file for TimeQuest to read in. This does not get used much because the user's .sdc files have usually been added to the project and are automatically read in by the Task menu's Read SDC File. This is covered in the next section. If the user wants explicit control to read in SDC files, this is how to do it.

Write SDC File… - This command is useful to see all the applied constraints in a text file. For example, the .sdc files created by Altera's DDR IP are long, parameterized, and hence very difficult to read. This command will write out a file showing all the constraints applied to a design. It's often too simplistic, in my opinion, but it's a nice feature. The one thing I do not recommend is to write out an .sdc that overwrites your own .sdc. This command should only be used to write to a test file, and if there are any constraints the user wants out of it, they should copy and paste them into their own .sdc file.

Reset Design - This command takes TimeQuest back to the point where a timing netlist has been created but before any .sdc files have been read in. It's very useful for the iterative method of creating constraints, whereby the user edits their .sdc file, resets the design, and then reads in the modified .sdc file. That being said, I usually access it from bottom of the Task menu on the left rather than the Constraints pull-down menu.

Getting Started - Timing Netlists and SDCs

There are three things TimeQuest must do before any analysis can be done:



Create Timing Netlist
Read SDC
Update Timing Netlist

They show up in the Tasks menu and will have a checkmark when completed. Note that the user does not have to double-click them individually. If they double-click on any report in the Tasks window, such as *Report Setup Summary*, then these 3 commands will automatically run. That means they will:

- Create a timing netlist based on the slow timing model.
- Read in all the .sdc files that have been added to the Quartus project, listed under Assignments -> Settings -> Add Files or TimeQuest Timing Analyzer, as well as any SDC commands embedded in the design files.
- Update the timing netlist, which is really just applying the timing constraints to the netlist so it can now be analyzed.

That's the default behavior for these three steps, but looking at them in more detail:

1) Create Timing Netlist. There are up to three timing models for an FPGA, which are explained here. When TimeQuest runs during a compilation, it will analyze the user's design against all available timing models, but when analyzing a design in the TimeQuest GUI, the user can only analyze one timing model at a time. Most setup and recovery failures are in the slow timing model, hold failures are in the fast timing model, and there is an occasional failure in just the slow 0° timing model.

The default is the slow timing model, since most failures occur in this model, but the user can go to the Netlist pull-down menu in TimeQuest and choose another timing model. They can also create a timing netlist for another speed grade, or they can create a Post-Map Netlist, which is based on the synthesis of the design but without any placement. (I would not use a

Post-Map Netlist for any serious timing analysis, but find this useful to edit an .sdc file and make sure it is doing what I want before running a full compile).

2) The next step is to read in the .sdc files, as well as any SDC constraints embedded in the HDL. The user can manually read in .sdc files from the pull-down menu Constraints -> Read SDC. To read in .sdc commands embedded in the HDL, type "read_sdc -hdl".

3) Finally the design must be updated. This is just applying all the SDC constraints to the physical database so that analysis can be done.

Notes:

- To start over, the user can go to the pull-down menu Netlist Delete Timing Netlist.
- To switch to a different netlist, the user can go to Netlist -> Set Operating Conditions, or access this command from the bottom of the Tasks menu.
- Finally, the task Reset Design is a great way to iteratively modify the .sdc commands and then re-analyze. This iterative method was described in the Getting Started section.

Major Reports

After compiling a design, the one command in TimeQuest I always run first is the macro *Report All Summaries*. This automatically runs the first three steps just shown as well as the five major reports:

Report Setup Summary

Report Hold Summary

Report Recovery Summary

Report Removal Summary

Report Minimum Pulse Width

Report Max Skew Summary

(It also runs *Report Clocks*).

The first four summary reports show each domain in the design, their slack and Total Negative Slack. Here is an example Setup Summary:

Summary (Setup)

	Clock	Slack	End Point TNS
1	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	0.770	0.000
2	tx_clk_ext	1.394	0.000
3	n/a	1.480	0.000
4	the_adc_pll|altpll_component|auto_generated|pll1|clk[2]	2.567	0.000
5	adc_clk_100_ext	2.609	0.000
6	the_system_pll|altpll_component|auto_generated|pll1|clk[0]	4.060	0.000
7	tx_clk	4.060	0.000
8	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	4.233	0.000
9	sys_clk	5.967	0.000
10	the_adc_pll|altpll_component|auto_generated|pll1|clk[0]	8.860	0.000
11	the_system_pll|altpll_component|auto_generated|pll1|clk[2]	9.303	0.000
12	adc_clk	10.938	0.000

As you can see, 12 clock domains are analyzed, where the worst slack is 0.770ns. All paths are grouped by their latch clock, so if a path has a different source and destination clock, it's the destination clock that it will be reported under for Summary reports. The End Point TNS is Total Negative Slack, which is the sum of negative slacks for each endpoint(if there are multiple failing paths to an endpoint, only the worst slack will be used). By itself I don't find TNS very useful, but when comparing to previous compiles of the same design is there some benefit. For example, Slack is always based on the single worst-case path in that domain, so if the user modifies code that is not the worst case path, the TNS may go up or down, signifying other paths got better or worse. To be honest, I don't find TNS all that beneficial though.

So why do I consider the Setup, Hold, Recovery and Removal Summaries the "Major Reports"? It's because they encapsulate every path in every domain that TimeQuest is analyzing and the fitter is trying to close timing on. I've seen too often where a user only looks at the Fmax Summary, which only covers paths within a domain. Or they might only run the Setup Summary, but have Recovery Paths that are failing and that the fitter is optimizing for at the expense of setup. These reports are really "the big picture", and hence it's important that users start at this high-level approach that shows them everything, and then work their way down with the command *report_timing*, which will be analyzed shortly.

The Minimum Pulse Width Summary analyzes structures that are capped at certain delays or frequencies. A good example is that a user might have a domain consisting of a single register feeding another register. If the data delay and clock skew were 1ns, then the summary setup report would state that the clock could run at 1GHz. The problem is device clock trees are capped at lower frequencies, as described in each device's handbook. So if the user tried to constrain this clock to 1GHz, it might pass the setup analysis but would fail the minimum pulse width check. Besides clock trees, other examples of structures that are capped are memory blocks, DSP blocks and I/O ports. All of these will show up in the Minimum Pulse Width check if the user tries to run them faster than they can handle. Note that the fitter generally can't do anything to improve Minimum Pulse Width checks, as it is a hard limit on a single portion of the device. Generally the user needs to either select a faster speed grade when possible, or lower the clock rate. The rate at which structures are capped should be in the device handbook.

The final report is Report Maximum Skew Summary. Unlike the other major reports, this one won't actually run in most designs because it only analyzes paths that have been constrained by *set_max_skew*. Since the majority of designs do not use this constraint, the majority of

designs will not have this reported. But any design that does have *set_max_delay* constraints, this is the final summary to state if they made timing or not. If the user wants more detail than the summary report, they can run:

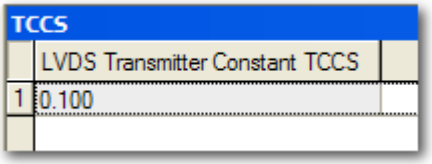
report_max_skew -detail full_path -panel_name "Detailed Max Skew"

Device Specific Reports

These reports are device specific and design specific. If your design does not use True LVDS blocks or Altera's DDR PHY(altmemphy or UniPHY), then they are not generally not relevant. If you are unsure if they apply to your design, simply double-click and see what happens. If they apply to your design, you will get a report. They will also be run in TimeQuest during a full compile, and any failures would show up there.

Report TCCS

This command reports Transmitter Channel-to-Channel Skew, which is only relevant on designs using True LVDS Transmitters. These are created with the altlvds megafunction, and must use the dedicated LVDS silicon. A full explanation of TCCS is given in each device's specific handbook, so please refer to that for timing diagrams. Here is an example TCCS report:



There is not much to this report. It doesn't say what I/O it is referring to, although this should be known by the user based on what outputs use True LVDS. The value is independent of timing model(a single value covers all models), and generally independent of device, package and speed grade within a family. This value can also be pulled from the datasheet.

The user does not need to enter any timing constraints on their True LVDS outputs, and if they do, those constraints will be ignored. This report does not have any pass/fail mechanism, it just states a value. Because of this, the user wants to make sure they get their clock/data relationship correct when setting up the altlvds block. For example, the TCCS value is the same whether clock and data are sent edge-aligned or center-aligned, yet obviously a receiver can only handle one of those relationships. I have never seen this be a problem, but think it's worth clarifying.

Report RSKM

The command reports Receiver Skew Margin, and is relevant on designs using True LVDS Receivers without Dynamic Phase Alignment (static timing analysis is not run on DPA, since that changes timing dynamically). A full definition with timing diagrams of RSKM, RCCS and the Sampling Window is given in each family's handbook. Here is a sample report:

RSKM

	RSKM	LVDS Period	Sampling Window	RCCS	Data Port	LVDS Channel Register
1	0.475	1.250	0.300	0.000	din[3]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_3~POS_CAP_DF
2	0.475	1.250	0.300	0.000	din[2]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_2~POS_CAP_DF
3	0.475	1.250	0.300	0.000	din[1]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_1~POS_CAP_DF
4	0.475	1.250	0.300	0.000	din[0]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_0~POS_CAP_DF

The report identifies which input ports it is analyzing, and makes use of input timing constraints, although in a minimal manner. If the user applies *set_input_delay -max* and *set_input_delay -min* constraints on the inputs, the values are used to determine RCCS. It takes the difference of the -max and -min values and applies that as the RCCS. The above report did not have any input constraints, so the RCCS is 0. After adding:

set_input_delay -clock sys_clk -max 0.100 [get_ports din]*
set_input_delay -clock sys_clk -min -0.050 [get_ports din]*

The report now looks like so:

RSKM

	RSKM	LVDS Period	Sampling Window	RCCS	Data Port	LVDS Channel Register
1	0.400	1.250	0.300	0.150	din[3]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_3~POS_CA
2	0.400	1.250	0.300	0.150	din[2]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_2~POS_CA
3	0.400	1.250	0.300	0.150	din[1]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_1~POS_CA
4	0.400	1.250	0.300	0.150	din[0]	...er:inst|altlvds_rx:ALTLVDS_RX_component|my_receiver_lvds_rx:auto_generated|rx_0~POS_CA

As can be seen, the RCCS went up by 150ps, which is the difference between the -max and -min values I applied. Note that the independent *-max* and *-min* values do not matter, as this report only looks at the difference between them. If I had entered a *-max 10.150* and *-min 10.000*, the difference would still be 150ps and I would have identical analysis. As such, this report does not analyze how well the clock is centered in the data eye and assumes the user has taken care of that when creating the altlvds receiver.

Personally, I do not see most users entering constraints, and just looking at the raw RSKM value and determining on their own if that is good enough. Since this value is a constant, this method works fine.

Report DDR

This command runs a timing analysis for designs using Altera's DDRx PHY, specifically the ALTMEMPHY or UniPHY. This analysis makes use of .tcl and .sdc files written out by the cores during generation, and should exist in the user's project directory. This is all explained in detail in the DDR IP documentation.

Report Metastability

I have to admit, I have not had a user concerned about metastability, and hence do not have real experience with these reports. I am not sure why this hasn't gained in popularity. My guess is that user's have gotten by this long without a good metastability analyzer, and hence feel it's probably not important. That's a shame because one nice feature of metastability problems is that, once identified, they are usually very easy to fix, by adding a little more timing margin on

the synchronization path, or adding another synchronization register. The difficulty with metastability has always been in identifying where it might be a problem in a real design. TimeQuest has the capabilities to do that analysis, giving Mean Time Between Failure(MTBF) values on individual synchronizers and across the entire design.

Note that MTBF analysis is not something you just turn on. The user must assist TimeQuest to make sure it analyzes the correct synchronizing registers, and if they want, tell TimeQuest the toggle rate of the data. (A reset signal may only toggle once a day, and hence is orders of magnitude less likely to suffer an MTBF failure than a data signal that is constantly changing). This is all detailed in the Quartus II handbook. Just search on www.altera.com for metastability, and select the Quartus II handbook link to Managing Metastability.

report_timing - If you only know one command…

I break this command out into its own section because 99% of my design analysis is done with this command. It really is the work-horse of TimeQuest and should be understood by every user. For starters, I recommend typing the following in TimeQuest:

r*eport_timing -long_help*

The *report_timing* dialogue box can be accessed from Report Timing.. in the Tasks menu on the left, or the pull-down menu Reports -> Custom Reports -> Report Timing. Both of these methods will pull it up "empty", whereby the user has to fill in what criteria they want to analyze. More often than not, *report_timing* is accessed by right-clicking on an existing report and selecting Report Timing. This uses *report_timing* as a diving tool, whereby they look at what domains are failing in the Summary reports and dive down with *report_timing* to get more detailed information.

This command is so important it is covered in the Getting Started section. Here are some more detailed notes on the command:

TQ_Analysis.tcl

When analyzing a project, I create a new Tcl file in my project directory called TQ_analysis.tcl. When analyzing paths I may want to analyze again, I copy my *report_timing* command out of the TimeQuest console and into TQ_analysis.tcl. This way I can access those commands in the future without having to go through the report_timing dialogue box. The most common case is for I/O interfaces. I might do something like:

report_timing -setup -npaths 100 -detail full_path -to [get_ports txout] -panel_name "s: * -> txout"*

report_timing -setup -npaths 100 -detail full_path -to [get_ports txout] -panel_name "h: * -> txout"*

This two commands analyze setup and hold to the output bus tx_out*. For inputs, I would use the -from option instead. Often an even better filter for I/O is to use *-from_clock* or -

to_clock. Since most I/O interfaces have a virtual clock created just for them, filtering on that clock automatically filters onto every I/O port associated with that clock.

Once TQ_analysis.tcl has been saved in the project directory, it can be accessed in TimeQuest from the Scripts pull-down menu. This allows easy access to analyze these specific paths anytime in the future without having to re-create the filters.

When creating a TQ_analysis.tcl file, it is especially important to pay attention to the *-panel_name*. This option specifies the name used for that report in Reports section. It often defaults to the generic name "Timing Report", and the user should be careful to change this to something more descriptive or else multiple calls of *report_timing* will just over-write existing reports with the same name. As a suggestion, I often start with what analysis is being done, using s: h: rec: rem: to represent setup, hold, recovery or removal. Next I use some shorthand to specify the path:

*-panel_name "s: * -> *KeyRAM*"*

This tells me the report does setup analysis on all paths with *KeyRAM* as the destination. This is my own syntax, and the designer should do whatever makes the most sense to them. One thing that most users don't know is that adding two pipe characters into their panel name will add hierarchy to their reports. So if I have a bidirectional bus called pci address, there are 4 things to analyze, setup and hold going off chip and setup and hold coming into the chip. Taking these four analysese:

r*eport_timing -setup -npaths 100 -detail full_path -to [get_ports {pci_address[*]}] -panel_name "PCI Address||s: * -> pci address"*

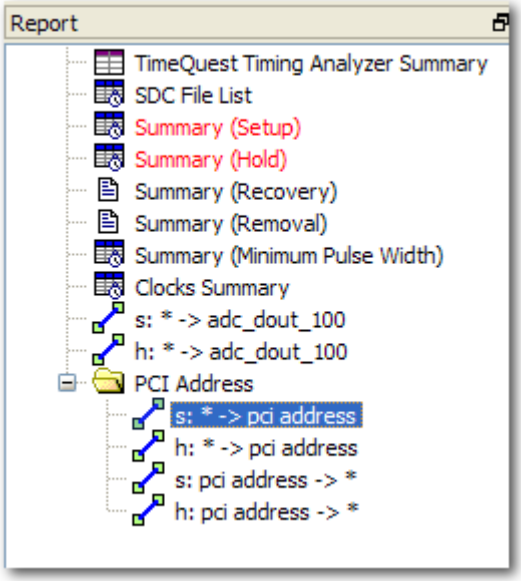
r*eport_timing -hold -npaths 100 -detail full_path -to [get_ports {pci_address[*]}] -panel_name "PCI Address||h: * -> pci address"*

r*eport_timing -setup -npaths 100 -detail full_path -from [get_ports {pci_address[*]}] -panel_name "PCI Address||s: pci address -> *"*

r*eport_timing -hold -npaths 100 -detail full_path -from [get_ports {pci_address[*]}] -panel_name "PCI Address||h: pci address -> *"*

Running these commands after Report All Summaries creates a Report table as shown on the right.

As can be seen, PCI Address is a folder that can be opened and closed, containing the four sub-reports within it. If your TQ_analysis.tcl script contains a lot of *report_timing* commands, this is a good way to help organize the results.



-false_path

The -false_path flag filters *report_timing* to only show paths that have been cut. These false paths are created in the .sdc files by either *set_false_path* or *set_clock_groups* commands. This flag is an extremely useful tool to look for paths that have either been incorrectly cut or signals that go between asynchronous domains without correctly being synchronized. This methodology is discussed in more detail in the Miscellaneous section Strategies for False Paths.

If the user forgets to constrain a clock, this option won't report paths to that clock. It only reports paths that have clock constraints and are then cut. To find clocks that aren't constrained to begin with, use Report Unconstrained Paths.

Path Filters

First off, I would suggest avoiding the -through option. This is not because it doesn't work, but the -through implies combinatorial logic, and combinatorial logic naming is always subject to the vagaries of synthesis. Endpoints are generally registers and I/O ports, which are much more reliable for matching the name in the original RTL. I'm not saying the -through filter should never be used, just that it should be avoided if the same thing can be accomplished with -from/-to.

The endpoint filters have many options. There are a slew of -fall_* and -rise_* options that I never use. These will limit your paths to only rising or falling edge transfers. The more generic -from/-to and -from_clock/-to_clock cover both rising and falling edges, which is usually fine. Users can filter on both clocks and endpoints, and only paths that meet both criteria will be reported. If an endpoint or clock is not specified, then all clocks or endpoints are allowed. So something generic like the following, which has no endpoint or clock filters, will list the worst 500 failing paths in the design, regardless of clock or endpoint:

r*eport_timing -setup -detail full_path -npaths 500 -pairs_only -panel_name "s: 500 worst paths"*

Note that I used the option *-pairs_only*. This option will limit the report to only 1 path between a pair of endpoints. Paths with large blocks of combinatorial logic can have many different routes between the endpoints, where the user really only cares about the worst case one. This may reduce the size of a timing report to something more readable. Note that *-pairs_only* is filtered during the display, so if 500 paths are found in the example above, *-pairs_only* most likely filter that to a smaller number.

Another good filter is *-nworst #*. I often run with *-nworst 1*. This will only show one path per destination and can reduce the number of failing paths reported by an order of magnitude. This often provides a much easier to read snapshot of the failing paths in a design. Be careful though, as it limits the information shown. For example, -nworst 1 might show only one path feeding a register, and that path might look like its placement was bad. Without the option *-nworst*, *report_timing* might show hundreds of sources feeding that register, which would explain why the critical path is forced to be spread out.

Datasheet Reports

Report Fmax

I am not a fan of this report. Fmax is only reported within a clock domain, and is therefore based solely on paths where the launch and latch clock are the same. Although the majority of paths meet this criterion, it is still a limited view of timing closure and a limited view of what the fitter is working on. Any paths between clock domains that are being analyzed and worked on by the fitter will be ignored by these reports.

My biggest complaint with report Fmax is that many customers rely on it rather than Setup Summary. When a design's critical paths are within a domain, then the Fmax and Setup reports will match. If the critical paths are between clock domains, those paths will show up in Setup Summary but not in the Fmax report. The only way to verify this is to run Report Setup Summary. Also, the Fmax report does not give detailed information on what paths it is using for analysis let alone what those paths look like. My feeling is that the sooner a user understands their setup reports, the better off they'll be and the sooner they'll realize they don't need the Fmax reports at all.

Report Fmax can be useful as long as the user understands it is only analyzing a subset of paths in the design. Many people prefer to talk about clocks in term of Fmax rather than slack, and this allows a quick reference point. That being said, it is generally not difficult to convert setup slack into Fmax.

Report Datasheet

Report datasheet will report the timing on your I/O ports in a device-centric manner, i.e. it reports Setup and Hold times on your input ports, Clock to Output and Min Clock to Output Times on your output ports, and Propagation Delays and Minimum Propagation Delays for input to output port connections with no registers in between. These reports have been requested so that users can describe their FPGA's timing with a fixed number, much like a device datasheet. This is a useful request, but has some flaws.

First, these reports are based on a specific place-and-route. If an output pin has a requirement that it get its data out in 6ns, and on a particular compile it is done in 5ns, Report Datasheet will say the Clock to Output is 5ns. That may be correct, but if the design is re-fit, it's possible the Clock to Output could get worse, up to 6ns, and there would not be any warnings or errors. From that perspective, I believe it makes more sense that the FPGA's I/O timing should be looked at based on its requirements, rather than the results of a single place-and-route.

Of course, I/O requirements are not made in a device-centric manner, which makes this more difficult. In reality, once a user understands the basics of I/O, it hopefully becomes clear how their requirements relate to these values. A more thorough discussion on device-centric versus system-centric timing can be found here. As can be seen, values like Tsu, Th and Tco make sense in many I/O cases, but there are a number of situations where they can't do complete analysis(such as source-synchronous interfaces) and a number of timing effects that they can't properly handle(like PLL phase-shifts and clock inversions).

The other issue I have with Report Datasheet is similar to my problem with Report Fmax, in that users rely on it to determine if their device met I/O timing. There are two major problems with this. First, Report Datasheet does not look at the user's requirements and therefore has no

pass/fail mechanism. Relying on this report for I/O timing would require the user to look at the numbers after EVERY compile and determine if they were good enough. Just as importantly, these reports do not give detailed path analysis. If the Clock to Output time on a port was 10ns but the user needed 8ns, there are no path details to tell them why their Clock to Output delay is so long. The user must go to their setup and hold reports to get these delays, and they will only get them if they constrain their I/O.

Much like Report Fmax, Report Datasheet does have its uses. Minimally they provide a quick glimpse of I/O timing. For slow interfaces, users often won't constrain them, and they may just occasionally look at these reports to make sure their values aren't significantly off from what they expect. It is important to note how this report deals with two major issues, clock inversion and PLL phase-shifts:

Clock inversion - The example I like to use with this is to think of a simple Clock to Ouput, where a clock comes into the FPGA and clocks data through an output register to the output port. Let's say the clock period is 20ns and after place-and-route, the Clock to Output is 7ns. Very straightforward. Now let's say everything is the same except the designer changes the output register so it is clocked on the falling edge. The data will now come out shifted by a half-period, 10ns, from when it was coming out before. Should the Clock to Output time be -3ns, 17ns, or 7ns? None of these actually seem right?

The answer is 7ns. So if two signals output ports were clocked on opposite edges of the clock, their data would come out at very different times, but their Clock to Output values would be the same. Luckily, the Datasheet has a column called Clock Edge. The following screenshot is from a design with an output bus called tx_out[7:0], where the upper four bits are clocked on the falling edge and the lower four bits on the rising edge. As can be seen, they have similar Clock to Output Times, but the lower bits are labeled with Rise while the upper bits are labeled with Fall:

15	⊟ tx_data[*]	tx_clk	6.286	6.265	Rise	tx_clk
16	tx_data[0]	tx_clk	6.213	6.185	Rise	tx_clk
17	tx_data[1]	tx_clk	6.112	6.077	Rise	tx_clk
18	tx_data[2]	tx_clk	6.286	6.265	Rise	tx_clk
19	tx_data[3]	tx_clk	6.247	6.235	Rise	tx_clk
20	tx_par	tx_clk	6.285	6.273	Rise	tx_clk
21	⊟ tx_data[*]	tx_clk	5.376	5.371	Fall	tx_clk
22	tx_data[4]	tx_clk	5.376	5.371	Fall	tx_clk
23	tx_data[5]	tx_clk	5.373	5.368	Fall	tx_clk
24	tx_data[6]	tx_clk	5.375	5.354	Fall	tx_clk
25	tx_data[7]	tx_clk	5.320	5.299	Fall	tx_clk

This column is easy to ignore, which is why it's worth pointing out.

PLL Phase-shifts - The second issue with device-centric timing concerns PLL phase-shifts. Let's start with the same example as above, whereby a clock with a 20ns period clocks data out and it takes 7ns. Let's say that it's going through a PLL now, so the Clock to Output time is still 4ns. (A PLL normally compensates for the clock tree delay and makes the Clock to Output timing better). Now let's start phase-shifting the PLL.

Let's start with a small phase-shift, say +/-500ps shift, and the Clock to Output times become 3.5ns or 4.5ns, respectively. This generally makes sense. Large shifts tend to be where things get confusing. If the user phase-shifts the clock +180 degrees of -180 degrees, the data will come out at the exact same time, but the +180 degree shift will give a Clock to Output time of 14ns and the -180 degree phase-shift will give a Clock to Output time of -6ns. Having a

negative Clock to Output does not generally make sense. In essence, it treats the phase-shift like a delay element on the clock path.

This is not like normal setup and hold analysis, where any user inserted phase-shift is not a delay, but actually effects the setup and hold relationships. In normal setup and hold analaysis, clocks are periodic functions, where a +180 degree phase-shift will be analyzed the same way as a -180 degree phase-shift. All in all, the way the Datasheet Report handles phase-shifts is how many users think, but it's important to understand the differences between its report and the real analysis being done for setup and hold.

Diagnostic

report_clocks

This command is the only diagnostic report that runs as part of Report All Summaries. This report nicely tells what the clocks look like after everything has been read in, including .sdc files, and hence what TimeQuest is using for analysis. This "view" can be much more straightforward of what's going on, rather than digging through RTL for clock names, or into .sdc files from IP vendors to see what clocks are created. It also clearly states the clock name, which can be cut and paste from for use in .sdc constraints or timing reports.

report_clock_transfers

I'm a big fan of this command. It gives an excellent report of how many paths exist between every pair of clocks, false paths included. If there are no physical connections between two clocks, they won't show up. Here's the report from a sample design:

Setup Transfers

	From Clock	To Clock	RR Paths	FR Paths	RF Paths	FF Paths
1	adc_clk	adc_clk	3	0	0	0
2	sys_clk	adc_clk	false path	0	0	0
3	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	adc_clk_100_ext	1	0	0	0
4	adc_clk	sys_clk	false path	0	0	0
5	sys_clk	sys_clk	3	0	0	0
6	the_adc_pll|altpll_component|auto_generated|pll1|clk[0]	the_adc_pll|altpll_component|auto_generated|pll1|clk[0]	1	1	1	0
7	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	5	1	1	0
8	the_system_pll|altpll_component|auto_generated|pll1|clk[0]	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	false path	0	0	0
9	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	false path	0	0	0
10	the_system_pll|altpll_component|auto_generated|pll1|clk[2]	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	false path	0	0	0
11	the_adc_pll|altpll_component|auto_generated|pll1|clk[2]	the_adc_pll|altpll_component|auto_generated|pll1|clk[2]	1	1	1	0
12	the_system_pll|altpll_component|auto_generated|pll1|clk[0]	the_system_pll|altpll_component|auto_generated|pll1|clk[0]	5	1	1	0
13	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	the_system_pll|altpll_component|auto_generated|pll1|clk[0]	1	0	0	0
14	the_adc_pll|altpll_component|auto_generated|pll1|clk[1]	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	false path	0	0	0
15	the_system_pll|altpll_component|auto_generated|pll1|clk[0]	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	1	0	0	0
16	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	the_system_pll|altpll_component|auto_generated|pll1|clk[1]	9	1	1	0
17	the_system_pll|altpll_component|auto_generated|pll1|clk[2]	the_system_pll|altpll_component|auto_generated|pll1|clk[2]	3	1	1	0
18	tx_clk	tx_clk	9	1	5	0
19	tx_clk	tx_clk_ext	5	4	0	0

This report gets generated for setup, hold, recovery and removal. I find Setup Transfers to be the most interesting. Note that RR Paths refer to paths where both registers are clocked on the rising edge. RF, FR and FF refer to different combinations of rise/fall clock transfers.

Finally, domains with false_path indicate that timing was cut between these clocks, either with a set_false_path between the clocks or in a set_clock_groups assignment. If the clock transfer is not cut at the clock level but some individual paths are cut, the number of paths between those clocks will be reported and the false paths will be included in that number. (E.g if there are 10 paths between two clocks, and 5 of them are cut with a *set_false_path*, this report will still state 10 paths exist.)

The user can right-click on any row and either do a Report Timing…, or Report False Path. These two commands will explicitly show the paths between those domains, with the number of paths and detail level specified by the user. Remember that Report False Path, which is just report_timing with the -false_path flag added to it, will only report paths that have been cut, either at the path level or at the clock level, so Report Timing and Report False Path will create a mutually exclusive list of all paths between those clocks made up of real paths and false paths.

Major domains that clock a lot of logic will have a LOT of paths listed, which is expected and not that useful of a number. Instead, it's the domains that have a small number of transfers, usually less than 100, that I find interesting. The first thing to ask is if the domains are related. If they are it's not a big deal, it just means a small number of paths send synchronous data. But if the clocks are asynchronous to each other and paths exist, is that expected? Quite often it is, and the paths will be inside an asynchronous FIFO, or perhaps a clock adaptor bridge inside SOPC Builder. But if the transfers are not expected, the user should investigate those paths and see if a mistake was made. Debugging incorrect transfers between asynchronous clocks is difficult in simulation, and extremely difficult in hardware, so being able to identify them in other ways can be useful. This is discussed more in the miscellaneous section on strategies for false paths.

There's a quick trick for getting rid of the "false path" description in this report. I will open my .sdc file and comment out the *set_clock_group* assignments, as well as any *set_false_path* assignments that are between clocks. I will *reset_design*, and then double-click Report Clock Transfers. This will read in the edited .sdc files that do not have any domains cut, and will report the number of paths between every domain. It may be helpful to take a screenshot of the original report or put it in a text file to compare which domains are really cut with this new report that shows how many paths exist between all domains.

Report Unconstrained Paths - report_ucp

This is an extremely important report, as it identifies unconstrained paths in the design. The most useful items it reports are unconstrained clocks and unconstrained I/O.

Unconstrained clocks start at the most basic, which is an input port that is used as a clock and does not have a clock constraint. After that are generated clocks, such as PLL outputs and transceiver outputs, which are usually covered by *derive_pll_clocks*, but would be missed if you're not using that command. Finally, ripple clocks, which occur when a register's output drives the .clk port of another register, need a *create_generated_clock* assignment or else they will show up as an unconstrained clock in this report. The one thing that will not show up is a

gated clock, i.e. when clocks go through purely combinatorial logic such as a mux. In these cases, the base clocks just pass through the structure, and hence it is not unconstrained, but could be constrained with a *create_generated_clock* at the mux output if the user wishes. See the section on clock muxes for more detail.

The unconstrained clock report is useful first off for any clocks the user forgot to constrain. If some clocks are unconstrained, they will be optimized for area during synthesis and the fitter will not try optimize paths within this domain. The user may say that's all right, as the domain may be very slow, but remember that hold violations can occur on the slowest of clock domains, i.e. 1Hz clocks can still fail a hold violation and still fail in hardware. More importantly, transfers between this domain and other domains will not be analyzed, which is another source for failure.

The second big use for unconstrained clocks is when a clock shows up that the user thought they constrained. The most likely scenario is from an error in the SDC file where their assignment did not take. The user should re-examine their assignment, as well as go back to the TimeQuest messages to see if a warning was issued when the assignment was processed.

Besides clocks, I/O ports are the next major portion of this report. Minimally, most users know they have not constrained all of their I/O, and hence this is a quick list of which I/O they missed. A constrained I/O port has one of the following constraints on it, *set_input_delay, set_output_delay, set_max_delay, set_min_delay,* or *set_false_path.* Note that output ports which send out a clock usually have a *create_generated_clock* assignment on them, but nothing else. These outputs show up in the unconstrained path report, although the comment section nicely states that it does have a clock assignment. I generally leave output ports sending clocks as unconstrained, although this will annoy some users, and some designs have requirements that all I/O are constrained. Adding a loose timing constraint would work around this:

```
set_max_delay 200.0 -to [get_ports clkout]
set_min_delay -200.0 -to [get_ports clkout]
```

The clock output won't be anywhere near those values, but these assignments will stop the port from showing up as unconstrained.

report_sdc

If your .sdc is straightforward, then this report won't do much more than report out what you put in. I find this most useful for complex constraints. For example, if the user's constraints are made of variables, it's sometimes helpful to see the constraint at its most basic level. For example, with an .sdc like so(I made up the values):

```
# CPU Specs:
set cpu_tco_max 6.123
set cpu_tco_min 3.434

# Board delays:
set cpu2fpga_max 0.877
set cpu2fgpa_min 0.488
```

set clk2cpu_max 1.455
set clk2cpu_min 1.011
set clk2fpga_max 1.505
set clk2fpga_min 1.074

 # Equations for CPU to FPGA:
set iMax_cpu [expr $clk2cpu_max + $cpu_tco_max + $cpu2fpga_max - $clk2fpga_min]
set imin_cpu [expr $clk2cpu_min + $cpu_tco_min + $cpu2fpga_min - $clk2fpga_max]

 # FPGA's inputs from CPU:
set_input_delay -max -clock cpu_clk_ext $iMax_cpu [get_ports i_cpu_*]
set_input_delay -min -clock cpu_clk_ext $imin_cpu [get_ports i_cpu_*]

That makes for a nicely descriptive SDC file, with the benefit of auto-calculating new requirements if the user changes the board delays or parameters of the external device. The only problem is that the final value isn't apparent without doing the math by hand. The user could add something like the following to their .sdc to echo the calculated value to the messages:

puts "iMax_cp => $iMax_cpu \n imin_cpu => $imin_cpu"

The problem is that you still have to find it in the messages. Running *report_sdc* allows the user to quickly find:

Set Input Delay

	SDC Command	Add Delay	Source Latency Included	Clock Fall	Flags	Clock Name	Reference Pin	Delay	Ports	Comments
1	set_input_delay	-add_delay			-max	[get_clocks cpu_clk_ext]		7.381	[get_ports i_cpu_addr[0]]	
2	set_input_delay	-add_delay			-min	[get_clocks cpu_clk_ext]		3.428	[get_ports i_cpu_addr[0]]	
3	set_input_delay	-add_delay			-max	[get_clocks cpu_clk_ext]		7.381	[get_ports i_cpu_addr[1]]	
4	set_input_delay	-add_delay			-min	[get_clocks cpu_clk_ext]		3.428	[get_ports i_cpu_addr[1]]	
5	set_input_delay	-add_delay			-max	[get_clocks cpu_clk_ext]		7.381	[get_ports i_cpu_addr[2]]	
6	set_input_delay	-add_delay			-min	[get_clocks cpu_clk_ext]		3.428	[get_ports i_cpu_addr[2]]	
7	set_input_delay	-add_delay			-max	[get_clocks cpu_clk_ext]		7.381	[get_ports i_cpu_addr[3]]	
8	set_input_delay	-add_delay			-min	[get_clocks cpu_clk_ext]		3.428	[get_ports i_cpu_addr[3]]	
9	set_input_delay	-add_delay			-max	[get_clocks cpu_clk_ext]		7.381	[get_ports i_cpu_addr[4]]	
10	set_input_delay	-add_delay			-min	[get_clocks cpu_clk_ext]		3.428	[get_ports i_cpu_addr[4]]	
11	set_input_delay	-add_delay			-max	[get_clocks cpu_clk_ext]		7.381	[get_ports i_cpu_addr[5]]	
12	set_input_delay	-add_delay			-min	[get_clocks cpu_clk_ext]		3.428	[get_ports i_cpu_addr[5]]	

Of course, the user could also run basic timing analysis on the path:

report_timing -setup -detail full_path -from [get_ports i_cpu] -panel_name "s: i_cpu*"*
report_timing -hold -detail full_path -from [get_ports i_cpu] -panel_name "s: i_cpu*"*

This will analyze the paths based on the constraints, and as discussed in correlating constraints to timing reports, the iExt delays would be 7.381ns in the setup report and 3.428ns in the hold report.

The *report_sdc* command is also useful if looking at constraints from an SDC created elsewhere, such as Altera's DDR2/3 IP cores. The user won't hand-edit machine-generated SDC files, but can use *report_sdc* to see what constraints were added.

Report Ignored Constraints - "*report_sdc -ignored*"

Ignored constraints will always produce a warning in TimeQuest's messages, which is useful, but often ignored by the user. I find this panel very useful to manage ignored constraints and try to get them down to as few as possible.

Note that ignored constraints are not always a problem. I have seen designs with various parameters that add/remove large sections of code depending on the build configuration. That code might have a lot of assignments, say multicycles and false paths, which are ignored when that block of code is not in the design. But unless the reason is well understood and accepted, ignored constraints should be cleaned up by the user. I also think they should be taken care of early on, rather than as a final design clean-up. The reason is that an ignored constraint often causes other problems that are difficult to debug.

A common example is when a user's *set_clock_groups* command has errors and is ignored, whereby all their asynchronous clocks become related, analyzed, and fail timing. The designer spends time analyzing a bunch of failing paths with impossible requirements, finally realizing they should not have been analyzed in the first place, and then going back to the TimeQuest messages to find why a constraint was ignored. If the user checked this report first, the problem would have been found much more quickly.

A more serious situation is when a user has multicycles or false paths within a domain that are ignored. The fitter might actually be able to close timing on those paths, so they don't show up as failures, but because they compete with real paths, those real paths suddenly get less-than-ideal placement. Without looking at the Ignored Constraints report, the user may never know of this problem and spend days/weeks trying to optimize timing through other methods, always assuming their exceptions were working.

And be aware that exceptions which are working might stop working midway through a project. One of the most common issues is when a hierarchy path changes, and hence the node names to everything beneath it have changed. If the assignments use full path names, they will no longer take. The hierarchy may change due another designer making a modification. It might be due to a different naming convention for generate statements. It may be due to regeneration of IP. All of these might occur without the user thinking to check if their .sdc constraints are still valid.

Recommendation: When possible, strive to get your design's Ignored Constraints report as close to having no ignored constraints as possible. The benefit is that if anything changes, new Ignored Constraints should be easily identifiable, and the user can fix the problem up front rather than debugging the secondary effects of an ignored constraint.

check_timing

This report was created by the TimeQuest group to look for common mistakes they see. Some of them are covered in other reports, such as unconstrained clocks or I/Os, and some give warnings in the messages, such as the PLL cross-check. These checks are not saying something is wrong, and if the user knows what they are doing there are many conditions where they would purposely design something that is flagged by check_timing. These checks are mostly stating that the design is doing something uncommon, and so the user might want to verify what they are doing is correct.

These checks are not documented very well. When they flag a possible issue, they give a quick description of the problem which is often clear enough, but in a few scenarios can leave the user scratching their head. I'll try to address as many as I can:

Virtual_clock - This flag occurs when no virtual clocks are found, which is generally a bad thing, since they are the basis of I/O constraints and really the first step for creating *set_input_delay* and *set_output_delay* constraints, as described in the Getting Started section.

This flag also occurs when a virtual clock is created, but not used in any constraints. Naturally if it's never used, there isn't any point in creating it, and so something may be wrong.

No Input Delay/No Output Delay - This check is not saying that the I/O are unconstrained, just that they don't have a *set_input_delay* or *set_output_delay* assignment on them. If they have a *set_false_path* assignment, then I consider that more than enough since you're explicitly saying the I/O should not be constrained. If the design has only *set_max_delay* and *set_min_delay* constraints, then it is not the official methodology for constraining I/O, but fine for users who understand what they're doing. A section on using these constraints for I/O is found here, while a section comparing the two methods is covered here.

I have found this check get flagged on True LVDS I/O, which generally do not need these constraints, as they get analyzed by Report TCCS and Report RSKM, or in the case of DPA Receivers, don't get reported at all. This is a case where the check needs to be analyzed by the user. Yes, True LVDS ports might not have input/output delay constraints, but they are also not needed.

Generated_IO_delay - This check occurs when the user has a *set_input_delay* or *set_output_delay* assignment whose -clock option uses a clock internal to the FPGA. The common scenario is when a new user enters the clock that drives the register inside the FPGA, such as the PLL that drives the input register:

set_input_delay -clock the_adc_pll|altpll_component|auto_generated|pll1|clk[0] -max 4.0 \ [get_ports din]*

As highlighted in red, the user is specifying an internal PLL clock for their set_input_delay constraint. New users often make this mistake and it is always wrong, since the analysis to the external register will use part of the FPGA's clock tree up to the PLL output, but that's it. Please look at the I/O timing section in getting started to understand.

Note that the name is a little misleading, since there is one common case where generated clocks work for I/O constraints. If the user has a *create_generated_clock* assignment on an output port to designate a clock being sent off chip, it is perfectly fine to use that clock for the -clock option of *set_input_delay* and *set_output_delay* constraints. This will not be flagged by check timing either. It is only when a *set_input_delay* or *set_output_delay's -clock* option uses a generated clock from inside the FPGA, such as a PLL output or ripple clock, will this get checked.

Partial Input/Output/MinMax Delay - These constraints usually come in pairs. For example, if the user does the following constraint, they have only applied the max delay analysis(i.e. setup analysis):

set_input_delay -max -clock cpu_clk_ext 6.0 [get_ports cpu_data]*

To be complete, the user should have a matching *set_input_delay -min* constraint to make sure the path is not too fast, which will be checked during hold analysis. This check occurs when a user has constrained a path, but only half of it.

Like many of the other checks, something getting flagged does not mean it is wrong. A user may override the default setup relationship on a path with *set_max_delay*, but keep the default hold analysis. This path would be flagged as only having a Partial Min-Max Delay constraint, which is true, but the user is all right with that since the min analysis done by the default hold relationship is what they want.

Partial Multicycle - This check occurs when a path has either a multicycle setup or hold, but not the other. If the user is trying to open the window, then this check is very useful, as the path will have a positive hold requirement and the design may become unroutable as it adds delay to meet that requirement, where the necessary multicycle -hold would fix the problem. If they are trying to shift the window, then the default hold relationship is all right and this check can be ignored. To get rid of the check, the user could add a matching multicycle hold that mimics the default hold:

set_multicycle_path -setup -from [get_clocks clk_a] -to [get_clocks clk_b] 2
set_multicycle_path -hold -from [get_clocks clk_a] -to [get_clocks clk_b] 0

The first constraint shifts the window. The second constraint is unnecessary since it mimics the default hold relationship, but it would prevent this timing check from being flagged.

PLL Cross Check - PLL's are configured based on how they are instantiated in the design, not what the timing constraints say. So if the user creates a PLL that has a 10ns input and creates a 5ns output, then physically the PLL will be configured for that. But if the user applies a timing constraint stating the input is 8ns, and *derive_pll_clocks* says the output is 4ns, there will be a PLL Cross Check flag. There will also be a warning in the messages.

The ability to have different settings is useful for a user who may want to run timing analysis at different rates without re-generating the PLL and creating a new image. In the example above, they may be curious if they can meet an 8ns input if they move up a speed grade, so they would *create_timing_netlist* for the faster speed grade and modify the SDC for the faster requirement, and just run TimeQuest to see the results. But if they are actually going to production, they need to regenerate the PLL with the new settings to ensure the correct bandwidth, VCO, and other internal settings are chosen for optimal performance.

Input Delay assigned to Clock - Clocks coming into the FPGA generally have a *create_clock* assignment, but do not have any *set_input_delay* assignments, which are for data inputs. This check alerts they user they have put a *set_input_delay* constraint on a clock, which

is probably not what they want. Usually this stems from using too broad of a wildcard for the port name and mistakenly matching the clock port.

report_partitions

This command cycles through all the partitions and does timing analysis within a partition and between partitions. It very nicely gives the user a sense of which partitions have the most difficulty, and if there are any inter-partition problems.

Custom Reports

Report Timing

This is the most important command for analyzing designs, and was covered at the beginning of this section.

Report Minimum Pulse Width

This command is the analysis tool for diving into minimum pulse width failures. These were described earlier in this section.

Report False Path

This is *report_timing* with the flag *-false_path* added. It is described here.

Report Path/Report Net

Report_path does timing analysis on a path(between registers or I/O ports). The clock delays to those endpoints are ignored, and there is no requirement. Likewise *report_net* will report the delay of individual nets, independent of any requirements. I have never found a good use for these, and more often than not find users going to these reports because they don't understand setup and hold reports, and want to try to do an analysis on their own. Minimally, these reports are missing vital information. Clock skew is generally always important, and they do not model On-Die Variation because they do not know if you want the slow or fast sub-models. Generally I have not found anything these reports can do that *report_timing* could not do better.

I would suggest the TimeQuest beginner concentrate on *report_timing*, since that shows the entire analysis that will drive the fitter and determine if the design passes timing. One possible benefit is that these commands will run on paths without any timing constraints, while *report_timing* requires constraints. I would argue that if the user is interested in the timing of a path, it probably needs a requirement.

Report Exceptions

The Report Exceptions analysis goes through all the exceptions in a user's .sdc files, such as *set_false_path*, *set_multicycle_path* and *set_max/min_delay*, and reports the status of that constraint. It determines if it matched any paths, if it was partially or completely overridden by another exception, and can report timing on paths covered by that exception. I would suggest running *"report_exceptions -long_help"* to read the description, as well as running it, since that's the best way to understand what it's doing.

All-in-all it's a very cool report, especially for a design with lots of exceptions that are hard to keep track of. The downside is that it takes a long time to run on a design with a lot of exceptions, since each exception translates to a call of *report_timing*. It also has a report on every single exception, which can be a lot of information. Because of long run times and long reports to wade through, *report_exceptions* can be unwieldy for general purpose analysis, but can be very useful for the occasional clean-up analysis, or to be run on a specific portion of the design.

As a data point, I ran this on an EP4SGX230 design, 80% full that did not have any user created exceptions. It took about 45 minutes to complete, and found almost 300 exceptions from the IP being used(QDR II, Altlvds, Asynchronous FIFOs). So in this case, it's an increase of 45 minutes and the report isn't overly helpful since all the exceptions are packaged in the IP, and hence not overly debuggable. For example, a lot of exceptions from the QDRII core come up as partially overridden or invalid, but since the user did not write the core, they have to assume they are correct. If a user has a hierarchy with a lot of user generated exceptions, it might be worthwhile to use the -to option to filter on that hierarchy.

Report Skew and Report Max Skew

These constraints report skew. Note that *report_skew* is a back-end tool for analyzing skew, in that the user specifies the endpoints they want to analyze. It is not a constraint but a back-end reporting tool. This is most useful for setting up an analysis, and then converting that analysis to a *set_max_skew* constraint in the .sdc file. Once that is done, *report_max_skew* will report the skew on all *set_max_skew* constraints. Skew constraints are discussed in more detail in the *set_max_skew* constraint section.

Report Bottlenecks

Run *"report_bottleneck -long_help"* in TimeQuest to get more information. I believe this command is similar to one in primetime, and used by ASIC designers to analyze timing problems. The premise is that just looking at long lists of paths based on their endpoints can leave the designer looking at the wrong things. Bottleneck is analysis of combinatorial nodes that have many critical paths going through them(with critical being defined by the -metric option).

This can be useful when the endpoints might not look like a pattern, such as various control signals going through a cloud of logic and fanning out to multiple hierarchies, and so looking at critical paths based on endpoints may show many paths that seem unrelated, where

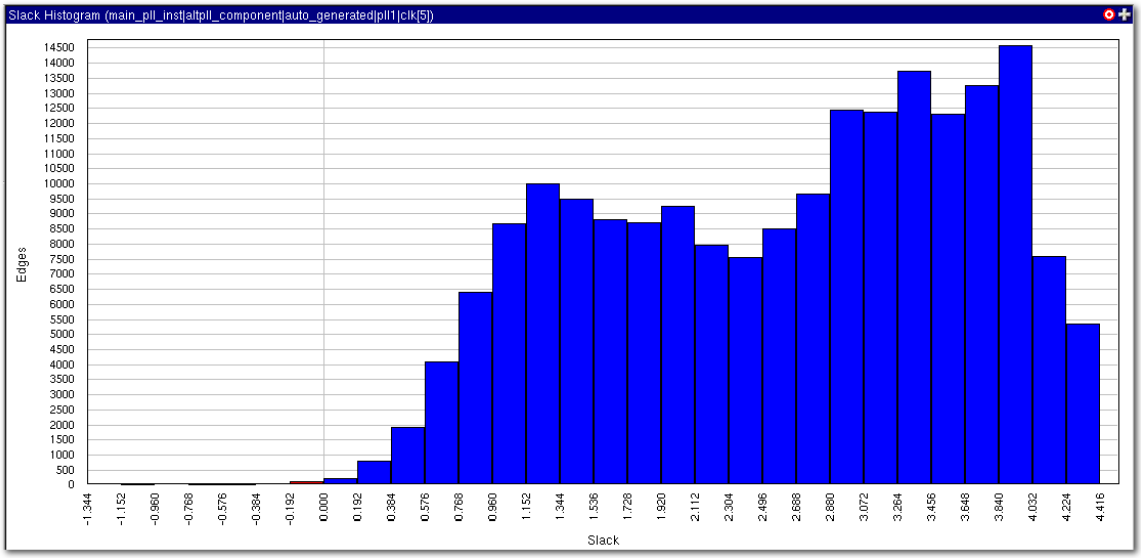
report_bottleneck would identify the bottleneck. Even when the endpoints are common, if they go through multiple hierarchies, the bottleneck may not be apparent.

Technically, this all sounds very good. In practice, I have not had this report help me in identifying something I couldn't determine from *report_timing*. Just as importantly, having identified a critical combinatorial node, it can be difficult to relate that back to the RTL, and even more difficult to determine an actionable fix for the problem.

The *report_bottleneck* command might be a useful tool for analyzing a design, but in general is probably not the first place to look.

Create Slack Histogram

This command gives a histogram of all paths within a domain and what their slack is. It's a nice way to show thousands or paths quickly:



First note that the vertical axis is based on Edges, not paths. An edge is a timing point in the design, generally a combinatorial or register output. It makes for a good metric instead of paths, since a single long-hop could create thousands of failing paths, yet it's really only one bad node placement.

I find this report somewhat interesting, but not very useful in determining next steps for timing closure or design optimization. I tend to think this makes for a better marketing slide than for a true analysis tool. One other thing to note is that the fitter concentrates on the worst case paths in a design. Let's take the slack histogram above and say the domain had a 1ns tighter requirement, so everything with a slack less than 1ns would be in red. This would mean tens of thousands of thousands of edges would be failing. But note that the fitter will really spend most of its time optimizing the most critical paths in a domain, since they determine its slack and how fast it can run, while other edges might get better timing if the fitter spent more time on them. By fixing the most critical paths in a design(a code change, a timing exception, etc.), the user might find less critical paths get fixed too.

Macros

Macros are essentially custom scripts that make use of existing commands. They are not direct commands, but instead special calls. For example, double-clicking **Report All Summaries** will actually run the following command within TimeQuest:

qsta_utility::generate_all_summary_tables

Note that this can be run by the user and can be put into the user's own Tcl analysis files.

Report All Summaries

I recommend all new users run this macro, and discuss it in more detail at the beginning of this section.

Report Top Failing Paths

A quick way to get the failing paths in all domains. The major downside to this report is that it defaults to just summary details. To get the full path details and analysis, the user must right-click on them and select Report Timing. I find it easier to run Report All Summaries and then right-click Report Timing on the domain of interest, whereby I can set the -detail level to something more robust like path_only or full_path, and thereby get low-level details on each failing path. Still, this macro delivers a nice snapshot of failing paths in the design.

Report All I/O Timings

I/O constraints describe a register outside of the FPGA, so the I/O analysis are just register-to-register paths just like internal paths. As such, they are reported with all the internal paths, which can be annoying for the user who wants the I/O broken out separately. This macro nicely does that, but has the downside of only giving a summary report, whereby the user still has to right-click Report Timing on a given path to get more details.

This command will not report I/O that are not constrained.

Note that there is nothing overly special about this macro, and the user could do their own Tcl script to achieve similar results, allowing them to modify settings such as the number of paths, the detail level, or write out to a text file. For example, the following does the same but all reports have *-detail full_path.*

```
report_timing -setup -npaths 1000 -detail full_path -from [get_ports *] \
    -panel_name "Report I/O Timing||Inputs to Registers (Setup)"
report_timing -hold -npaths 1000 -detail full_path -from [get_ports *] \
    -panel_name "Report I/O Timing||Inputs to Registers (Hold)"
report_timing -recovery -npaths 1000 -detail full_path -from [get_ports *] \
    -panel_name "Report I/O Timing||Inputs to Registers (Recovery)"
report_timing -removal -npaths 1000 -detail full_path -from [get_ports *] \
    -panel_name "Report I/O Timing||Inputs to Registers (Removal)"
report_timing -setup -npaths 1000 -detail full_path -to [get_ports *] \
    -panel_name "Report I/O Timing||Registers to Outputs (Setup)"
```

*report_timing -hold -npaths 1000 -detail full_path -to [get_ports *] *
-panel_name "Report I/O Timing||Registers to Outputs (Hold)"
*report_timing -setup -npaths 1000 -detail full_path -from [get_ports *] -to [get_ports *] *
-panel_name "Report I/O Timing||Registers to Registers (Setup)"
*report_timing -hold -npaths 1000 -detail full_path -from [get_ports *] -to [get_ports *] *
-panel_name "Report I/O Timing||Registers to Registers (Hold)"

Report All Core Timing

Similar to *report_timing* on a specific domain, except I/O paths are excluded. As with the other macros, this only reports summary detail and the user must right-click Report Timing on a specific row to get details on that path.

Create All Clock Histograms

This macro is a shortcut to create histograms for every clock domain, rather than using *create_slack_histogram* to make them one by one. The pros and cons of histograms are discussed on the individual command.

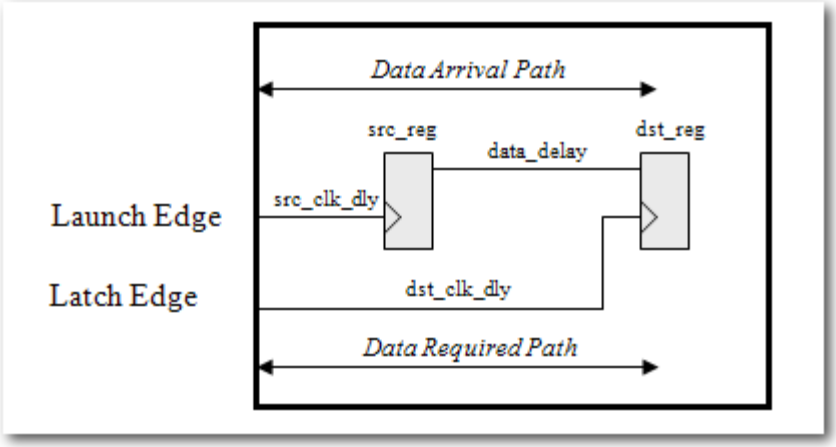
Section 5: Timing Models

Why Timing Models are Important

Most critical paths designers deal with are in the core of their FPGA, and usually within the same clock domain, which is using a global. In such cases, users tend to ignore the clock delays and just analyze the data path. With such an analysis, all they need to know is how slow the data path can be(setup analysis) and how fast the data path can be(hold analysis). In such circumstances, we naturally want the timing models to be as accurate as possible, but if they're too pessimistic, that's all right. Our design may not run as fast, but at least we know our design will work.

The basic parameters that affect these delays are Process, Voltage and Temperature, which will be referenced as PVT throughout. Process accounts for the variation in different devices coming out of the FAB. They are all tested and binned into speed grades, but still have variation over that process. Voltage covers the fact that the voltage will vary over time, which directly causes the device to run faster or slower. A higher voltage makes it run faster. Finally there is Temperature, whereby a lower temperature makes the devices run faster. Generally, these three variables are lumped together and considered as two data points, the slow timing model (what's the slowest my design will run on the slowest device that met the speed grade, at the lowest voltage in spec and the highest temperature in spec) and the fast timing model(what's the fastest my design will run on the fastest device, highest voltage and lowest temperature). The general analysis is that, as long as the design passes timing under these two data points, it is ready to go.

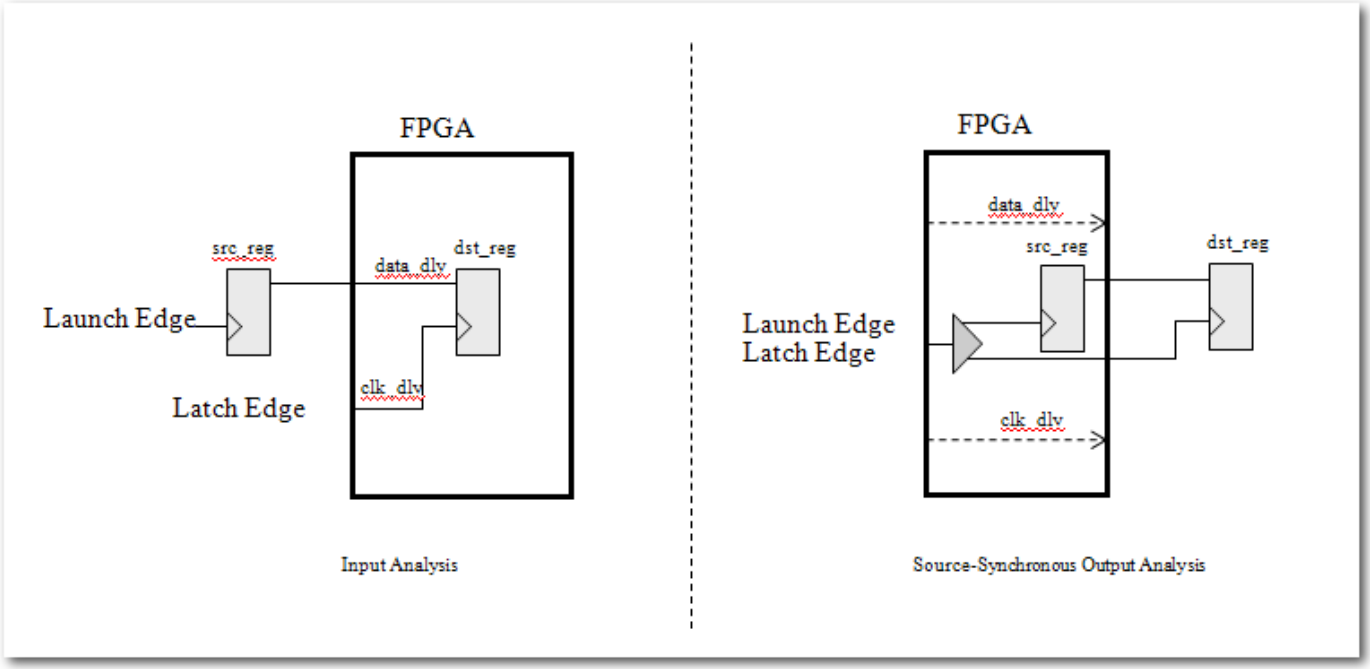
That's the "simple view" of static timing analysis. Although accurate at a high level, there are many more issues at play. It might be worth reviewing the Basics of setup, hold, recovery and removal. The important point to recognize is that timing analysis is not just how long or short a path it is. Instead, it is the measure of the Data Arrival Path compared to the Data Required Path:



For setup analysis, we want the Launch Edge to get data to the dst_reg before the Latch Edge. For hold analysis, we want it to get there after the Latch Edge. As a result, we have two signals racing against each other, and don't want to measure their extremes by themselves,

but want to measure them in relation to each other. This is where the difficulties of dealing with real silicon come into play. Even if the device was at the slow corner(worst process, high temperature, low voltage), not all paths within the device will actually be at that worst case delay. For example, our data arrival may be at that slowest point, but our data required path may actually be running a little faster. This could be due to localized variations in PVT within a single device, rise/fall variation in the transistors, PLL jitter, and a myriad of other issues that occur in silicon. If we model the Data Required Path as if it were at the worst case corner, then we're being optimistic compared to how real silicon behaves, and might pass static timing analysis while our device fail in the field. For setup analysis, we really want the slowest Data Arrival Path compared to the fastest possible Data Required Path that could occur simultaneously.

This is exacerbated in cases where the data arrival path and data required path have matching delays, such as inputs, where the data and clock path are often quite similar, or on source-synchronous outputs, where the user wants the clock and data delays to be aligned as close as possible:

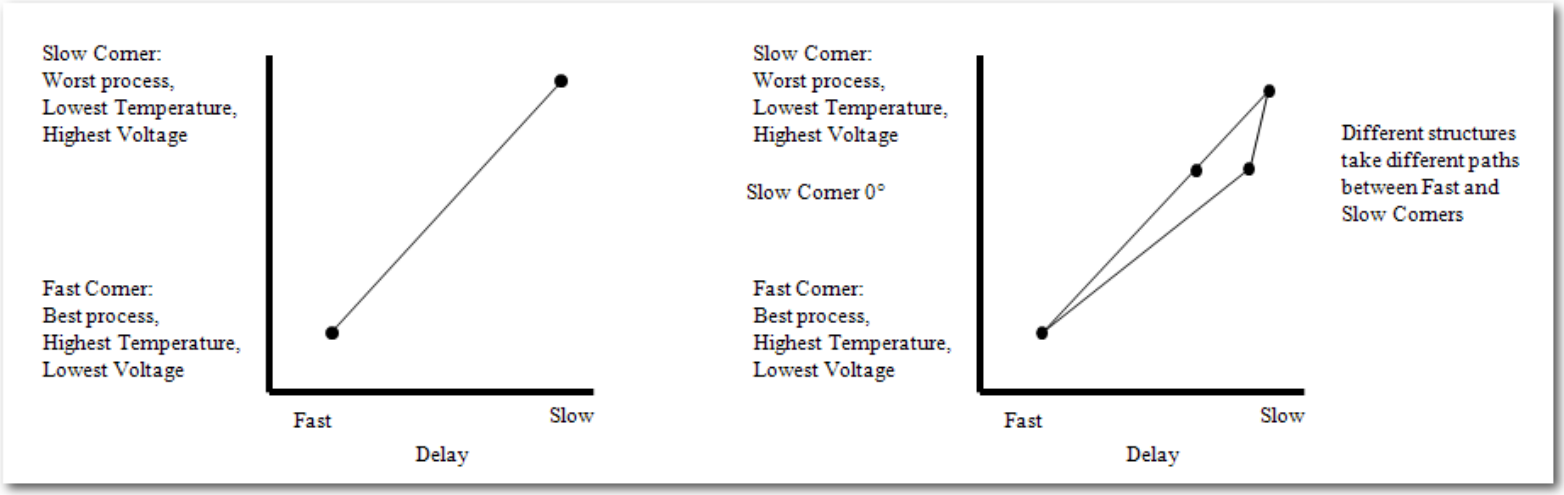


In these cases, the two signals race each other the whole way, and without accounting for these variations, we get inaccurate results. An excellent example is when users just look at Tcos for a source-synchronous output. The Tco is a spec for the worst case delay to the output without relation to anything else. If the user compares the Tco of a clock and data leaving the FPGA, they might find them to be within tens of picoseconds of each other. This is true, since that is the worst case possible delay for each path. But when trying to see how different they can really be in hardware, all sorts of other phenomenon such as on-die Variation, temperature inversion, rise-fall variation, will make the variance much higher, possibly adding hundreds of picoseconds.

Luckily TimeQuest has ways to account for all of these. Let's see how:

Timing Models

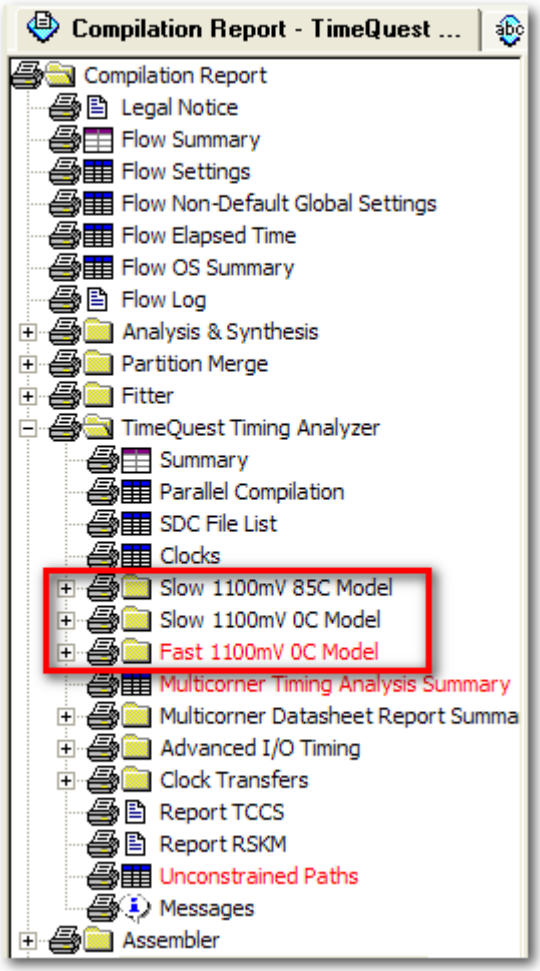
Historically there have always been two timing models for sign-off, a slow corner and a fast corner. This is shown in the diagram on the left:



With two models, the assumption is that everything tracks between those two models in a similar manner. For example, if a transistor is at the 50% point between the fast and slow models, than all transistors in the die are at 50%, all wires, etc. The line between the Fast and Slow Corners does not have to be straight, it just has to be bounded by the Fast and Slow points, and all structures in the FPGA must move in unison. The first point is true, in that the points at the Fast and Slow Corners are the extremes, but not all structures move between these points in unison. This issue has been exacerbated by a process called temperature inversion, whereby delays can actually decrease with rising temperatures.

The end result is that a third timing model was added, called the "Slow 0° Corner". The graph above on the right shows two different paths between the Slow and Fast, whereby different structures take different routes between the Fast and Slow Data points. (The graph's only purpose is to show different paths are possible. I completely made up the magnitude and shape of the lines). The important point is that, when comparing two paths, which is what static timing analysis does, structures may following different paths between the slow and fast corners. The slow 0° model is meant to capture that analysis. I have seen real designs that pass both the fast and slow corners, but fail in this middle model.

Analysis of this third model is done by default, but can be found under Quartus II's pull-down menu of Assignments -> Settings -> TimeQuest Timing Analyzer -> Enable Multi-Corner Timing Analysis During Compilation. It can be studied in TimeQuest after Updating the Timing Netlist by going to the pull-down menu of Netlist -> Set Operating Conditions. The results of all three timing models can be found in the Compilation Report under TimeQuest:



Uncertainty

The clocks described in a user's .sdc are perfect, where their edges repeat down to the exact picoseconds. In reality, clocks aren't perfect for various reasons. A big one is that PLLs can add jitter. The *set_clock_uncertainty* constraint allows users to add uncertainty, but the user doesn't know what uncertainty is inside the FPGA. Luckily the *derive_clock_uncertainty* command will determine this uncertainty based on the user's design. In general, this is the only constraint necessary to cover clock uncertainty.

Rise/Fall Variation

Transistors have different rise and fall times, which TimeQuest uses in its analysis. Note that it's not just rise and fall times, but the cell delays are based on what type of edge comes in and what kind of edge comes out. This means there are 4 unique delays, RR, FF, RF, and FR, where the first letter is the edge coming in, and the second letter is the edge going out. Here's a screenshot from a timing report where the RF column is highlighted:

Path #1: Setup slack is -8.003

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location
1	0.000	0.000				
2	⊟ 0.000	0.000				
3	0.000	0.000	R			
4	4.150	4.150	R	iExt	1	PIN_AV18
5	⊟ 23.309	19.159				
6	4.150	0.000	RR	IC	1	IOIBUF_X97_Y0_N63
7	4.684	0.534	RR	CELL	4	IOIBUF_X97_Y0_N63
8	13.211	8.527	RR	IC	1	MLABCELL_X92_Y44_N36
9	13.626	0.415	RR	CELL	6	MLABCELL_X92_Y44_N36
10	15.089	1.463	RR	IC	1	MLABCELL_X92_Y47_N18
11	15.470	0.381	RF	CELL	1	MLABCELL_X92_Y47_N18
12	15.939	0.469	FF	IC	1	MLABCELL_X92_Y47_N8
13	16.125	0.186	FR	CELL	2	MLABCELL_X92_Y47_N8
14	16.275	0.150	RR	IC	1	MLABCELL_X92_Y47_N0
15	16.653	0.378	RF	CELL	6	MLABCELL_X92_Y47_N0
16	18.538	1.885	FF	IC	1	MLABCELL_X85_Y32_N26
17	18.945	0.407	FR	CELL	1	MLABCELL_X85_Y32_N26
18	19.095	0.150	RR	IC	1	MLABCELL_X85_Y32_N36
19	19.510	0.415	RR	CELL	1	MLABCELL_X85_Y32_N36
20	19.822	0.312	RR	IC	1	MLABCELL_X88_Y32_N4
21	19.893	0.071	RF	CELL	1	MLABCELL_X88_Y32_N4
22	20.247	0.354	FF	IC	1	MLABCELL_X85_Y32_N32
23	20.475	0.228	FR	CELL	2	MLABCELL_X85_Y32_N32
24	22.757	2.282	RR	IC	1	DDIOOUTCELL_X98_Y0_N40
25	23.309	0.552	RF	CELL	1	DDIOOUTCELL_X98_Y0_N40

Rise/fall differentiation is pretty straightforward by itself, but becomes much more complex when analyzing multiple rise/fall elements in a row, in which case unateness comes into play.

Unateness

Rise/fall variation by itself creates delay values that are inaccurate. For example, let's say we had a chain of 3 AND gates followed by 3 OR gates, and wanted to determine the slowest delay through this structure. (Yes, the FPGA fabric is really made of LUTs, but for this example I'm looking at logic as if it were gates). Now let's say the slowest delay through the AND gate is RR, i.e. rising in to rising out, and the slowest delay through the OR gate was FF. If we summed each gate's worst delay we would get a worst case total delay that is impossible. The reason is that, if the AND gates have a RR edge propagating through them, there is no way for the OR gate to get a falling edge coming through them. In reality, this structure only has two possible delays through it, all falling edges or all rising edges, but no combinations of RF or FR.

Both an AND gate and OR gate are positive unate. This means a rising edge in will always create a rising edge out, and a falling edge in will always create a falling edge out. So only RR and FF models are necessary.

A NOT gate and NAND gate are negative unate, in which a falling edge in creates a rising edge out, and vice versa.

An XOR gate is non-unate, in which case all combinations are possible.

Potive unate = RR, FF

Example: AND gate, OR gate

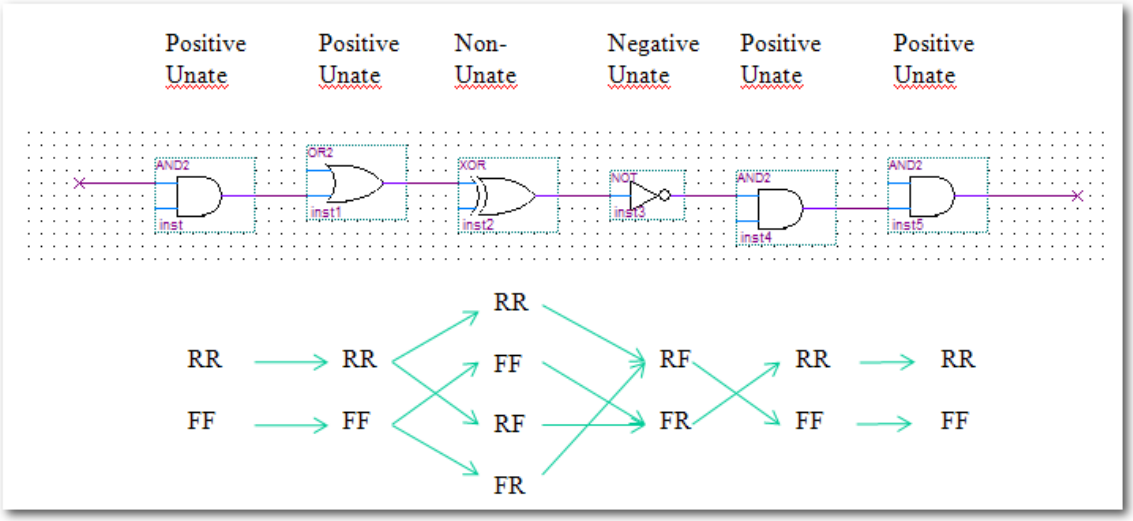
Negative unate = RF, FR

Examples: NAND gate, NOR gate, NOT gate

Non-Unate = RR, FF, RF, FR

Examples: XOR gate

The key to properly analyzing unateness is to determine what is possible through a circuit. Let's look at another series of gates:



I only showed valid delays for each gate. The first AND gate, for example, is positive unate and can only be RR or FF. From there, the arrows show the only possibly transitions, so the RR out of the first gate can only drive the RR of the next OR gate and cannot drive the FF. So if we wanted the slowest possible path, we would need to find the path that gives the longest delays. Likewise, the same has to be done for the fastest possible path.

For clocks delays, the analysis is restricted by how the register is clocked. If it's clocked on the rising edge, then TimeQuest will only analyze paths that result in a rising edge at the register.

This can be confusing at first, but it's important to note is that the user does not have to do anything for this, TimeQuest analyzes unateness behind the scenes and the correct edges are automatically used during timing analysis. The only reason this is discussed is so the user understands what is going on under the hood.

On-Die Variation

On-Die Variation, or ODV, is the principle that all paths within a die do not track with each other exactly. Note that this is not the same as the different timing models, which are used to analyze different macro-conditions, specifically process, voltage and temperature. ODV occurs at a given timing model's PVT, and measures the amount of variation that can occur at that macro point. I think of ODV as being a sub-timing model. So when doing timing analysis at the Slow Corner, there is a fast and slow sub-model. All three corners have these sub-models. The best way to explain it is to show it:

setup: inst8 -> inst9

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	5.967	..._CrossClocks|inst8	..._CrossClocks|inst9	sys_clk	sys_clk	10.000	-0.062	0.819

Path #1: Setup slack is 5.967

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 3.219	3.219					clock path
3	3.219	3.219	R				clock network delay
4	⊟ 4.038	0.819					data path
5	3.319	0.100		uTco	1	FF_X40_Y47_N23	cross_domain:NoPLL_CrossClocks|inst8
6	3.319	0.000	FF	CELL	1	FF_X40_Y47_N23	NoPLL_CrossClocks|inst8|q
7	3.571	0.252	FF	IC	1	FF_X40_Y47_N21	NoPLL_CrossClocks|inst9|asdata
8	4.038	0.467	FF	CELL	1	FF_X40_Y47_N21	cross_domain:NoPLL_CrossClocks|inst9

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	10.000	10.000					latch edge time
2	⊟ 13.157	3.157					clock path
3	13.126	3.126	R				clock network delay
4	13.157	0.031					clock pessimism
5	10.137	-3.020					clock uncertainty
6	10.005	-0.132		uTsu	1	FF_X40_Y47_N21	cross_domain:NoPLL_CrossClocks|inst9

hold: inst8 -> inst9

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	0.575	..._CrossClocks|inst8	..._CrossClocks|inst9	sys_clk	sys_clk	0.000	0.062	0.712

Path #1: Hold slack is 0.575

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 3.126	3.126					clock path
3	3.126	3.126	R				clock network delay
4	⊟ 3.838	0.712					data path
5	3.226	0.100		uTco	1	FF_X40_Y47_N23	cross_domain:NoPLL_CrossClocks|inst8
6	3.226	0.000	RR	CELL	1	FF_X40_Y47_N23	NoPLL_CrossClocks|inst8|q
7	3.462	0.236	RR	IC	1	FF_X40_Y47_N21	NoPLL_CrossClocks|inst9|asdata
8	3.838	0.376	RR	CELL	1	FF_X40_Y47_N21	cross_domain:NoPLL_CrossClocks|inst9

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					latch edge time
2	⊟ 3.188	3.188					clock path
3	3.219	3.219	R				clock network delay
4	3.188	-0.031					clock pessimism
5	3.208	0.020					clock uncertainty
6	3.263	0.055		uTh	1	FF_X40_Y47_N21	cross_domain:NoPLL_CrossClocks|inst9

Above we see *report_timing* run on the exact same path, the left side showing the -setup analysis and the right side showing the -hold analysis. Both of these are taken at the slow corner.

Note that most data paths that go through multiple levels of logic have multiple paths between the same register. When doing a setup analysis the slowest path shows up first, but when doing a hold analysis the shortest path shows up first. That alone will cause vary different results. In this particular case, there is only one path through a single LUT, so I am comparing the exact same path in both the setup and hold analysis.

For setup analysis, we want the slowest possible Data Arrival Path compared to the fastest possible Data Required Path. For hold analysis, we want the exact opposite. If you look at lines 7 and 8 of the Data Arrival Path, you find that the incremental delay on the left(setup) is slower and the delay on the right(hold). This is because TimeQuest uses different sub-timing models. For setup, the Data Arrival Path uses the Slow Corner, slow sub-timing model. For hold, the Data Arrival Path uses the Slow Corner, fast sub-timing model. This causes a difference of 100ps on this short path.

Note though that this is not all due to different sub-models. TimeQuest is also choosing different rise/fall options, as we have already discussed. The setup analysis chose FF, while the hold analysis chose RR. That's because the datapath will have both conditions traveling through it, and we want the worst possible case to make sure timing can be met.

On the other hand, the clock delays are only rising edge, since both registers are rising edge triggered. So looking at line 3 of Data Arrival Path, the network delay is slower for setup analysis than for hold analysis. This is purely from modeling On-Die Variation. Similarly, on

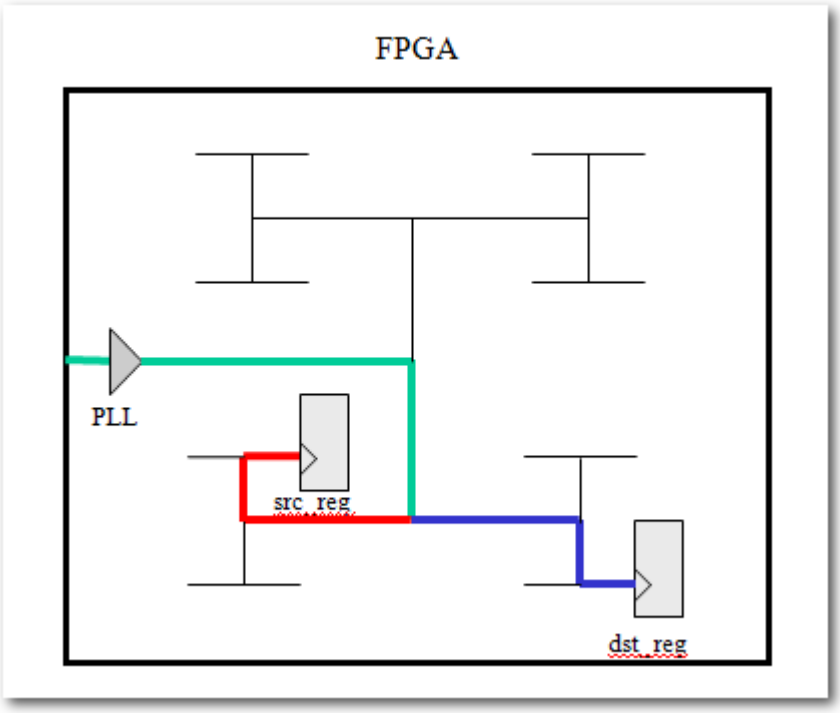
the Data Required Path, the clock network delay is faster for setup analysis than for hold analysis. ODV works in both directions, and so we must always choose the worst case model for our particular analysis. (The fact that clocks can't be both slow and fast will be covered in the next section, Common Clock Path Pessimism.)

Once again, this is all taken care of for the user underneath TimeQuest's hood, and there is nothing they need to do. The reason it's worth knowing is to understand why timing numbers on the same path may look different under different analysis. It also should help in the user's confidence when doing timing analysis.

As already stated, On-Die Variation is a real phenomenon. Without it, the timing models would be overly optimistic, and the hardware could fail on a design that passes static timing analysis. But one thing TimeQuest does not do in its models is account for locality. Two output ports right next to each other will have the same on-die variation in their analysis as two outputs on opposite sides of the device. In reality, locality does play a factor that is not accounted for. Without it, the current models are overly pessimistic though, meaning if they pass timing the hardware will only work better, but it can make timing closure more difficult. (The only place I have seen it be a problem is on source-synchronous outputs not using the True LVDS blocks, where ODV can make the timing seem quite bad, and in theory, accounting for locality could make them better.)

Common Clock Path Pessimism

As just discussed, On-Die Variation makes use of a slow and fast sub-model within each major timing model. This accounts for slight variations in the die, and is important since we are timing signals that race against each other. But in many signals, part of the source clock delay and destination clock delay are identical, i.e. they are fed by the same clock, and until it splits, there can be no On-Die Variation. Common Clock Path Pessimism removes any on-die variation for the common part of the clock. Let's look at a simple schematic:



In this example, the clock comes into the FPGA, through a PLL, onto the global clock tree, and at some point splits in different directions, one path feeding the src_reg and the other path feeding the dst_reg. Before the split, there is no on-die variation because it's the same path, and a single path can't vary from itself.

Setup and hold analysis will not see it this way. For setup, the entire Data Arrival Path, which includes the source clock delay of green and red lines, will be analyzed completely in the slow sub-model, and the Data Required Path, which includes the destination clock delay of the green and blue lines, will be analyzed with the fast sub-model. This can be shown in the following screen-shot:

Setup: inst2 -> inst3

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock
1	4.060	domain:inst5|inst2	domain:inst5|inst3	...nerated|pll1|clk[0]

Path #1: Setup slack is 4.060

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	F...	Location
1	5.000	5.000				
2	⊟ 5.501	0.501				
3	5.000	0.000				
4	5.000	0.000			1	PIN_N3
5	5.000	0.000	RR	IC	1	IOIBUF_X53_Y22_
6	5.753	0.753	RR	CELL	2	IOIBUF_X53_Y22_
7	8.001	2.248	RR	IC	1	PLL_R2
8	1.774	-6.227	RR	COMP	3	PLL_R2
9	1.774	0.000	RR	CELL	1	PLL_R2
10	3.255	1.481	FF	IC	1	CLKCTRL_G11
11	3.420	0.165	FF	CELL	10	CLKCTRL_G11
12	5.119	1.699	FF	IC	1	FF_X1_Y21_N21
13	5.501	0.382	FR	CELL	1	FF_X1_Y21_N21
14	⊞ 6.120	0.619				

Data Required Path

	Total	Incr	RF	Type	F...	Location
1	10.000	10.000				
2	⊟ 10.332	0.332				
3	10.000	0.000				
4	10.000	0.000			1	PIN_N3
5	10.000	0.000	RR	IC	1	IOIBUF_X53_Y22_
6	10.748	0.748	RR	CELL	2	IOIBUF_X53_Y22_
7	12.940	2.192	RR	IC	1	PLL_R2
8	6.539	-6.401	RR	COMP	3	PLL_R2
9	6.539	0.000	RR	CELL	1	PLL_R2
10	7.956	1.417	RR	IC	1	CLKCTRL_G11
11	8.100	0.144	RR	CELL	10	CLKCTRL_G11
12	9.681	1.581	RR	IC	1	FF_X1_Y21_N23
13	10.041	0.360	RR	CELL	1	FF_X1_Y21_N23
14	10.332	0.291				
15	10.312	-0.020				
16	10.180	-0.132		uTsu	1	FF_X1_Y21_N23

Hold: inst2 -> inst3

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship
1	0.362	inst14	inst16	...enerated|pll1|clk[0]	...enerated|pll1|clk[0]	0.000

Path #1: Hold slack is 0.362

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	F..	Location	Element
1	0.000	0.000					launch edge time
2	⊟ -0.147	-0.147					clock path
3	0.000	0.000					source latency
4	0.000	0.000			1	PIN_N3	sys_clk
5	0.000	0.000	RR	IC	1	...BUF_X53_Y22_N1	sys_clk~input|i
6	0.748	0.748	RR	CELL	2	...BUF_X53_Y22_N1	sys_clk~input|o
7	2.940	2.192	RR	IC	1	PLL_R2	the_system_pll|altpll_cor
8	-3.461	-6.401	RR	COMP	3	PLL_R2	the_system_pll|altpll_cor
9	-3.461	0.000	RR	CELL	1	PLL_R2	the_system_pll|altpll_cor
10	-2.044	1.417	RR	IC	1	CLKCTRL_G11	the_system_pll|altpll_cor
11	-1.900	0.144	RR	CELL	10	CLKCTRL_G11	the_system_pll|altpll_cor
12	-0.428	1.472	RR	IC	1	FF_X52_Y13_N25	inst14|clk
13	-0.147	0.281	RR	CELL	1	FF_X52_Y13_N25	inst14
14	⊞ 0.337	0.484					data path

Data Required Path

	Total	Incr	RF	Type	F..	Location	Element
1	0.000	0.000					latch edge time
2	⊟ -0.095	-0.095					clock path
3	0.000	0.000					source latency
4	0.000	0.000			1	PIN_N3	sys_clk
5	0.000	0.000	RR	IC	1	IOIBUF_X53_Y22_N1	sys_clk~input|i
6	0.753	0.753	RR	CELL	2	IOIBUF_X53_Y22_N1	sys_clk~input|o
7	3.001	2.248	RR	IC	1	PLL_R2	the_system_pll|altpll_cor
8	-3.226	-6.227	RR	COMP	3	PLL_R2	the_system_pll|altpll_cor
9	-3.226	0.000	RR	CELL	1	PLL_R2	the_system_pll|altpll_cor
10	-1.772	1.454	RR	IC	1	CLKCTRL_G11	the_system_pll|altpll_cor
11	-1.625	0.147	RR	CELL	10	CLKCTRL_G11	the_system_pll|altpll_cor
12	-0.103	1.522	RR	IC	1	FF_X52_Y13_N27	inst16|clk
13	0.196	0.299	RR	CELL	1	FF_X52_Y13_N27	inst16
14	-0.095	-0.291					clock pessimism
15	-0.075	0.020					clock uncertainty
16	-0.025	0.050		uTh	1	FF_X52_Y13_N27	inst16

The left side shows the setup analysis and the right side shows the hold analysis of the same path with the clock path broken out in more detail using *report_timing -detail full_path*. I highlighted a single cell delay, the input clock's IO buffer, which is in row 6 for all

four sections. Multiple lines have On-Die Variation for the common part of the clock tree. Line 14 of the Data Arrival Path there is a 291ps delay called clock pessimism. This represents the clock pessimism between the two sub-models that is not real, and basically adds back in the difference. Note that this 291ps is added to the Data Required Path, which makes it easier to meet setup timing, so common clock path pessimism removal is helping us close timing.

On the right side is the hold analysis, and you can see the numbers are reversed. The faster sub-model is used for a delay of 748ps on the Data Arrival Path, and the slower sub-model delay of 753ps is used on the Data Required Path. These differences occur throughout the clock tree, but line 14 subtracts 291ps of clock pessimism. This helps us meet hold timing. The bottom line is that common clock path pessimism always helps us meet timing, and hence is a good thing. Without it, we would be overconstraining this path by 291ps on both setup and hold.

The clock pessimism is a single line item, so it doesn't break out exactly where in the previous delays it is accounting for pessimism. This is most apparent in the clock tree, which in this example is CLKCTRL_G11. This global line has an IC delay of more than 1.4ns. Somewhere along that clock tree the clock will split, where part of it routes to the source register and part of it routes to the destination register. Only the part that is common will be accounted for by common clock path pessimism, but where that split occurs is not shown. (You could add do *report_timing -show_routing* to the path to get detailed routing info.)

In the end, there is nothing the user needs to analyze with common clock path pessimism removal, just make sure it's on since it helps close timing. It is on by default, and can be found under Assignments -> Settings -> TimeQuest Timing Analyzer. It's also useful to know why that line item is there, but there is really nothing the user has to do, as TimeQuest handles all the calculations.

Section 6: Quartus II and Timing Constriants

Section 7: Tcl Syntax for SDC and Analysis Scripts

Section 8: Common Structures and Circuits

PLLs

Dedicated Output
Clock Switchover

Transceivers

LVDS

Memory Interfaces

Clock Muxes

Ripple Clocks

Clock Enables

Section 9: Examples

Section 10: Miscellaneous

Strategies for False Paths

Analyzing Paths

Comparing set_input_delay/set_output_delay to Tsu/Th/Tco and min Tco