

TimeQuest для чайников.

Денис Шехалев
shdv@micran.ru
diod2003@list.ru

TimeQuest для чайников. Часть 1 (Введение)

В свое время фирма Altera, желая быть ближе к народу (удобство работы в ее софте это большой плюс), создала инструмент под названием Timing Analyzer. Было все просто, прописал тактовые частоты проекта и нажал run для анализа. Но проекты становились все сложнее, частоты все выше, а количество частот в проекте все больше, возможностей Timing Analyzer стало не хватать.

Нужен был более гибкий инструмент задания ограничений и их учета при имплементации проекта. И такой инструмент нашелся у синописиса. Это скрипты констрейнов для проекта. Для анализа проектов на основе данных скриптов и появился TimeQuest.

С этого момента произошло разделение Timing Analyzer на Classic Timing Analyzer и TimeQuest Timing Analyzer. Сама фирма Altera не рекомендует использовать Classic TA, потому как качество синтеза и анализа с этим анализатором хуже. Более того, для новых семейств ПЛИС возможности выбора Classic TA нет.

Очень часто от начинающих и не только альтероидов можно слышать реплики "Зачем мне TimeQuest, он такой сложный" или "Помогите разобраться, как он работает". Что могу сказать таким людям? Времена задания одной цифры в красивом ГУИ прошли, нужно учиться писать скрипты для TimeQuest. А если хотите еще большего хардкора, то поработайте с Xilinx и его скриптами %).

Итак, начнем с азов.

Все констрейны для TimeQuest пишутся в файле с расширением *.sdc. Altera не стала изобретать велосипед и выбрала Synopsys Design Constraint формат. Этот формат представляет собой TCL скрипт, в котором констрейны прописываются с помощью предопределенных команд. Как вы увидите, в этом нет ничего сложного.

Рассмотрим простой фпгашный проект HelloWorld:

```
module hello (input clk, output led) ;

    logic [31 : 0] cnt;

    always_ff @(posedge clk) begin
        cnt <= cnt + 1'b1;
        led <= cnt[31];
    end

endmodule
```

Как мы видим в этом проекте всего 2 порта: клок и светодиод. Нам нужно задать ограничения на тактовую частоту и описать, как анализировать вывод светодиода. Для этого создаем файл `hello.sdc`.

Клоки описываются с помощью команды `create_clock`, полный формат которой можно посмотреть в хендбуке на квартус. Пусть у нас используется генератор с частотой 10МГц.

```
create_clock -period 10MHz -name {clk} [get_ports {clk}]
```

Видите, ничего сложного: надо указать частоту, логическое имя клона и место источник этого клона (в данном случае это порт ПЛИС).

Так ли важно в этом примере, какова задержка вывода сигнала на светодиод? Естественно нет, поэтому нужно описать его как путь, который анализировать не нужно.

```
set_false_path -from [get_clocks {clk}] -to [get_ports {led}]
```

Я написал, что анализировать путь от триггера, который тактируется логическим клоком `clk`, до порта фпга `led` не нужно, ничего сложного.

На этом создание файла констрейнов, для одночастотного проекта типа "мигание светодиодом" закончено. Как видите, скрытых таинств нет %).

TimeQuest для чайников. Часть 2 (TimeQuest лицом к лицу)

В предыдущей части мы рассмотрели самый простой проект для ПЛИС, который только может быть. Перед тем как перейти к более сложным проектам надо освоить инструмент, о котором идет речь. Эта часть посвящена основам временного анализа с помощью TimeQuest.

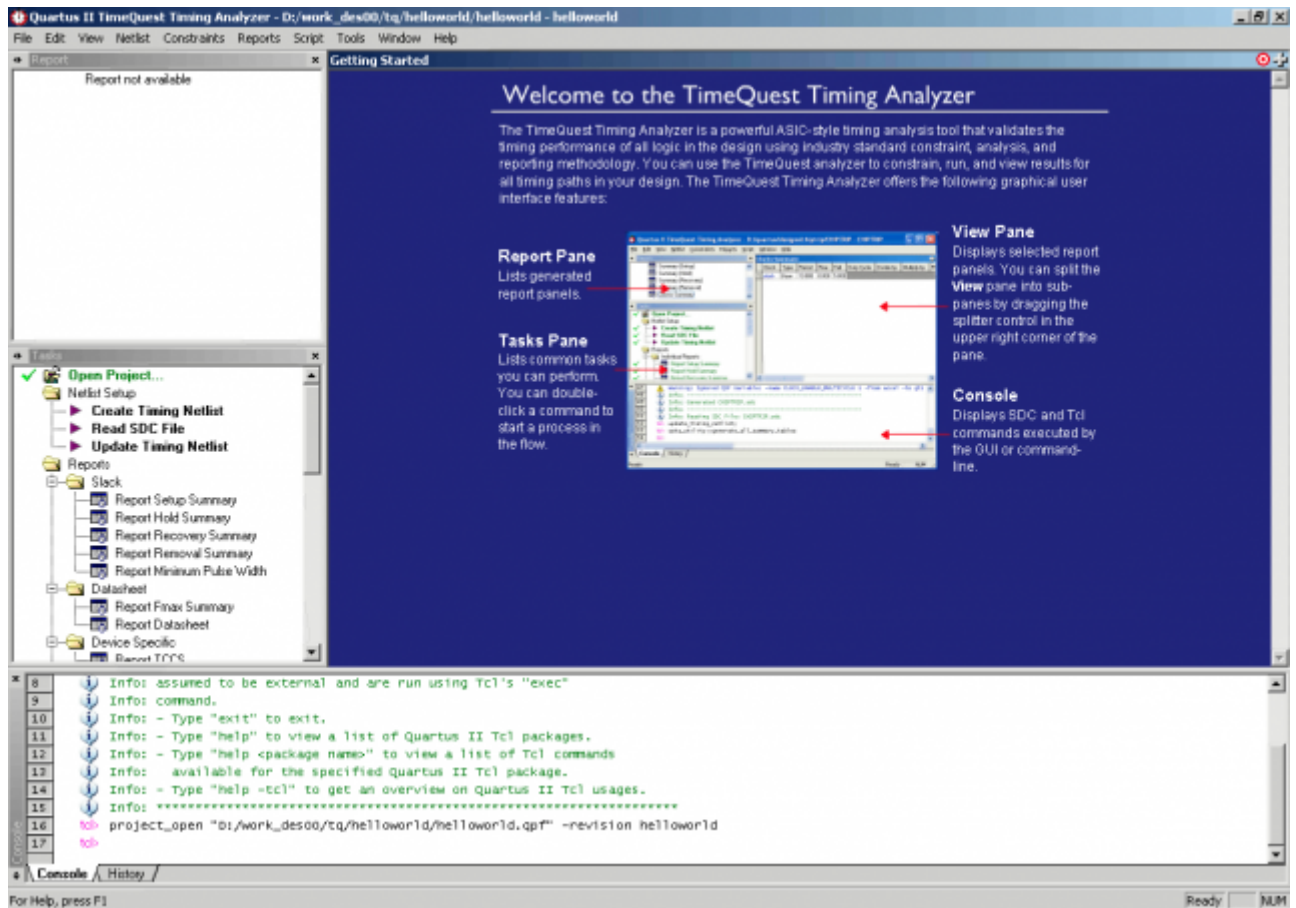
TimeQuest это программа для проверки выполнения временных ограничений, заданных в sdc файле констрейнов. Вот тут возникает первое правило TimeQuest а:

Если TimeQuest рапортует вам об отсутствии ошибок, то не надо обольщаться что все хорошо. Может быть, вы просто не задали часть временных ограничений.

Т.е. первое отличие от Classic TA в том, что TimeQuest проверяет только те ограничения, которые вы задали. Поэтому при написании sdc файла внимательно читайте предупреждения, которые он вам выдает.

С TimeQuest можно работать в двух режимах: графическом и консольном. В консольном режиме все sdc команды вводятся в консоли, а результаты могут быть просмотрены как в консоли, так и в специальных окнах. Но фирма Альтера позаботилась о пользователях и снабдила TimeQuest ГУИ интерфейсом, с помощью которого можно не зная полного синтаксиса sdc команд, работать с той же эффективностью. Это относится не только к командам анализа, но и к командам задания самих констрейнов.

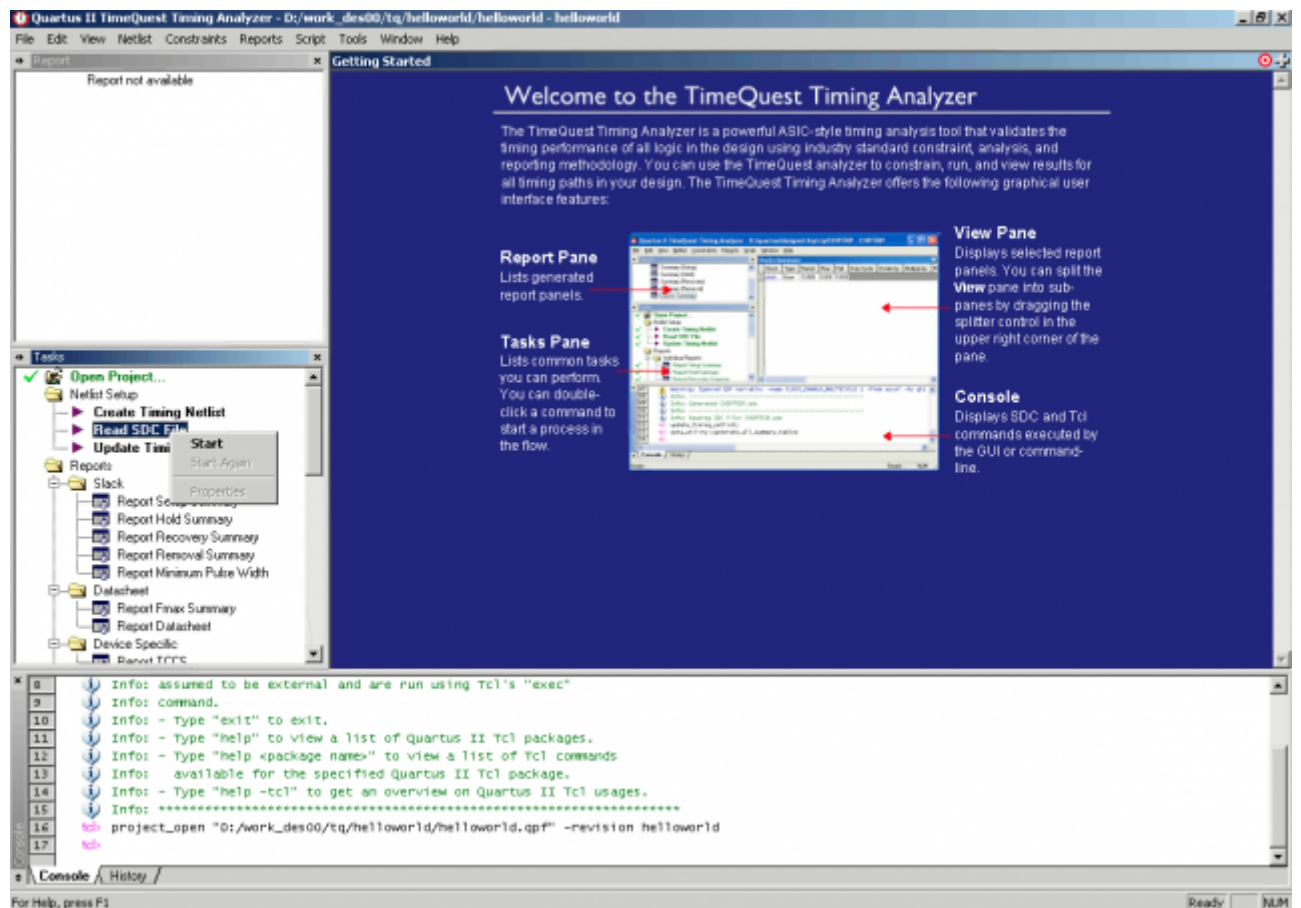
Итак, собираем наш проект HelloWorld. Запускаем TimeQuest. Видим несколько окон.



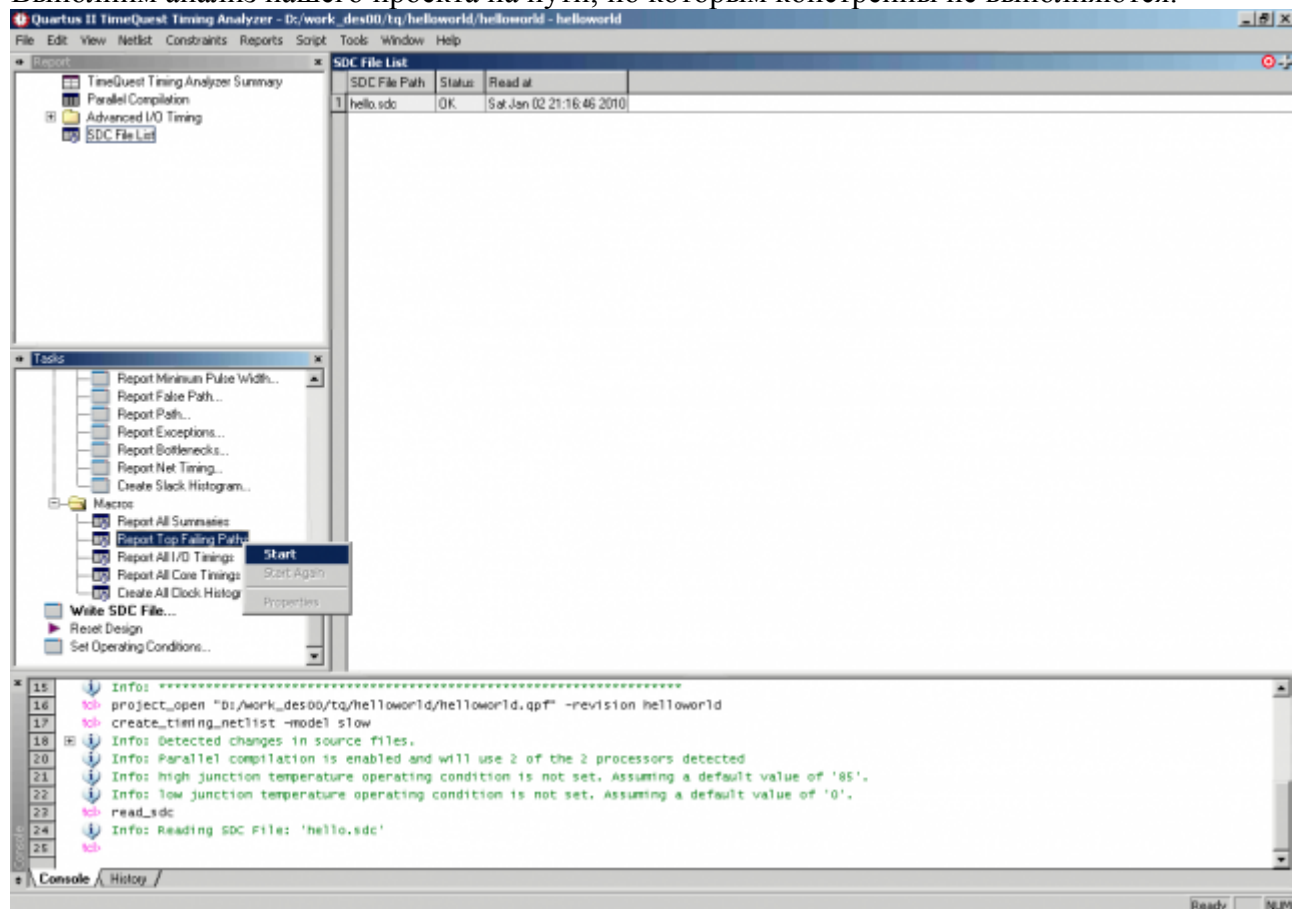
В окне Getting Started мы видим подсказку:

1. Tasks Pane - окно команд. В этом окне перечислены основные команды временного анализа.
2. Report Pane - окно логов. В этом окне отображаются закладки отчетов о временном анализе.
3. View Pane –окно отображения путей. В этом окне будут перечислены пути, которые использовались при анализе, а также отчеты об этих путях.
4. Консоль

Временной анализ возможен только по уже существующему нетлисту. Для анализа нужно загрузить нетлист и соответствующий ему sdc файл. С помощью правой кнопки мыша загружаем их.



Выполним анализ нашего проекта на пути, по которым констрейны не выполняются.



В консоли мы видим, какие команды выполнялись и их результат.

Info: Reading SDC File: 'hello.sdc'

update_timing_netlist

Critical Warning: The following clock transfers have no clock uncertainty assignment. For more accurate results, apply clock uncertainty assignments or use the derive_clock_uncertainty command.

Critical Warning: From clk (Rise) to clk (Rise) (setup and hold)

qsta_utility::generate_top_failures_per_clock "Top Failing Paths" 200

Info: No fmax paths to report

Info: No fmax paths to report

Info: No failing paths found

Разберем по полочкам, что все это значило. Для анализа TimeQuest применил загруженные в него констрейны к нетлисту (команда *update_timing_netlist*). Потом он обнаружил, что перечисленные констрейны не позволяют ему достоверно проанализировать нетлист и выдал соответствующее предупреждение. Затем выполнил анализ путей, по которым констрейны не выполняются. Как мы видим, таких путей нет (еще бы в таком то проекте на частоте 10МГц) и с этой точки зрения все хорошо.

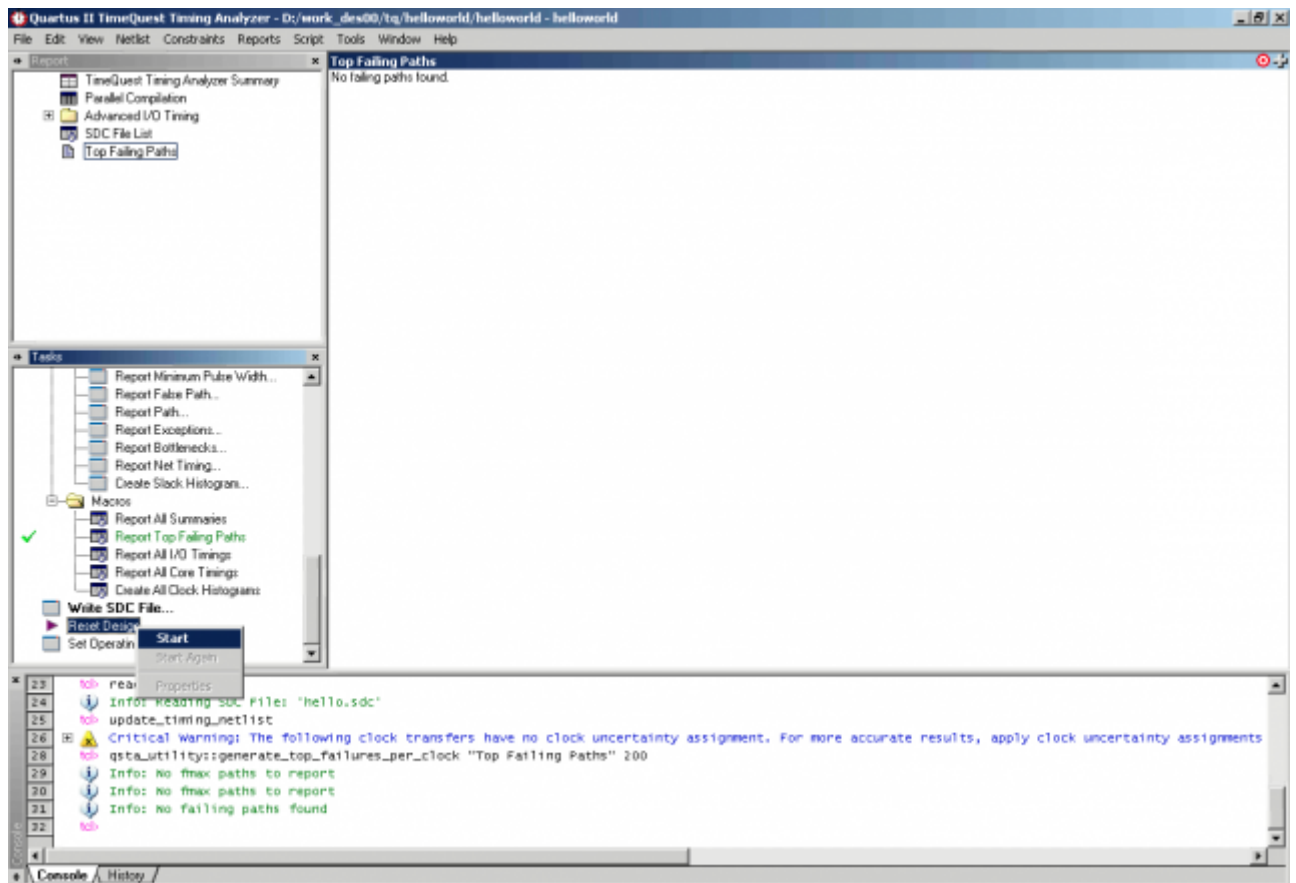
Но надо вылечить предупреждение. Для этого нужно изменить файл констрейнов. Дописываем недостающую строку. Теперь файл hello.sdc выглядит так:

derive_clock_uncertainty

create_clock -period 10MHz -name {clk} [get_ports {clk}]

set_false_path -from [get_clocks {clk}] -to [get_ports {led}]

Нужно перезапустить временной анализ. В принципе можно заново выполнить команду Read SDC File, но в этом случае поверх уже существующих констрейнов будут прописаны новые. И может случиться переопределение/схлестывание констрейнов. Поэтому я рекомендую делать такие вещи через полный сброс и повторный запуск анализа. Делаем сброс.



и запускаем анализ снова. В итоге нет ни предупреждения, ни ошибок.

В принципе не обязательно сбрасывать проект, наложить нужный констрейн можно и прямо с консоли или через меню constraints. Но для начинающих я рекомендую описанный выше способ. Также помните, что все констрейны, вводимые через меню или консоль, не сохраняются в sdc файле до тех пор, пока вы не выполните команду Write SDC File. Но я обычно пишу sdc файл ручками и инициализирую анализ заново.

Теперь давайте немного сломаем наш проект. Поставим самый медленный циклон 3 и в sdc файле напомним:

```
create_clock -period 300MHz -name {clk} [get_ports {clk}]
```

Результат анализа.

Quartus II TimeQuest Timing Analyzer - D:\work_des00\tq\helloworld\helloworld - helloworld

File Edit View Netlist Constraints Reports Script Tools Window Help

Report

TimeQuest Timing Analyzer Summary

- Parallel Compilation
- Advanced I/O Timing
- SDC File List
- Top Failing Paths
 - Setup: clk

Tasks

- Report Minimum Pulse Width...
- Report False Path...
- Report Path...
- Report Exceptions...
- Report Bottlenecks...
- Report Net Timing...
- Create Slack Histograms...
- Macros
 - Report All Summaries
 - Report Top Failing Paths
 - Report All I/O Timings
 - Report All Core Timings
 - Create All Clock Histograms
- Write SDC File...
- Reset Design
- Set Operating Conditions...

	Slack	From Node	To Node	Launch Clock	Latch Clock
1	-0.855	cnf[0]	cnf[31]	clk	clk
2	-0.730	cnf[2]	cnf[31]	clk	clk
3	-0.632	cnf[1]	cnf[31]	clk	clk
4	-0.600	cnf[5]	cnf[31]	clk	clk
5	-0.586	cnf[4]	cnf[31]	clk	clk
6	-0.499	cnf[3]	cnf[31]	clk	clk
7	-0.415	cnf[0]	cnf[30]	clk	clk
8	-0.351	cnf[5]	cnf[31]	clk	clk
9	-0.295	cnf[5]	cnf[31]	clk	clk
10	-0.269	cnf[5]	cnf[25]	clk	clk
11	-0.239	cnf[0]	cnf[29]	clk	clk
12	-0.197	cnf[7]	cnf[31]	clk	clk
13	-0.149	cnf[10]	cnf[31]	clk	clk
14	-0.146	cnf[1]	cnf[30]	clk	clk
15	-0.123	cnf[5]	cnf[25]	clk	clk
16	-0.114	cnf[2]	cnf[29]	clk	clk
17	-0.093	cnf[5]	cnf[27]	clk	clk
18	-0.084	cnf[2]	cnf[30]	clk	clk
19	-0.053	cnf[18]	cnf[31]	clk	clk
20	-0.051	cnf[5]	cnf[31]	clk	clk
21	-0.016	cnf[1]	cnf[29]	clk	clk
22	-0.002	cnf[12]	cnf[31]	clk	clk

Console

```
5.6 Info: Reading SDC File: 'hello.sdc'
5.7 Info: Clock uncertainty calculation is delayed until the next update_timing_netlist call.
5.8 update_timing_netlist
5.9 Info: Deriving Clock Uncertainty
6.8 qsta_utility::generate_top_failures_per_clock "Top Failing Paths" 200
6.9 Info: Report Timing: Found 22 setup paths (22 violated). Worst case slack is -0.855
7.6 Info: No fmax paths to report
7.7 Info: No fmax paths to report
7.8
```

На сленге это называется что-то вроде «все в слаках»/«сплошные слаки». Т.е. запас (в данном случае по *tsetup*) выбран и его не хватает для работы на нужной частоте. В зависимости от того, что является источником slack его можно побороть либо констрейнами (это предмет следующих тем), либо изменением дизайна.

Для определения узкого места нужно посмотреть подробный временной отчет о пути.

Quartus II TimeQuest Timing Analyzer - D:\work_des00\tq\helloworld\helloworld - helloworld

File Edit View Netlist Constraints Reports Script Tools Window Help

Report Setup: clk

Slack	From Node	To Node	Launch Clock	Latch Clock
1 -0.855	cnt[0]	cnt[31]	clk	clk
2 -0.730	cnt[2]	cnt[31]	clk	clk
3 -0.632	cnt[1]	cnt[31]	clk	clk
4 -0.500	cnt[5]	cnt[31]	clk	clk
5 -0.586	cnt[4]	cnt[31]	clk	clk
6 -0.495	cnt[3]	cnt[31]	clk	clk
7 -0.415	cnt[0]	cnt[30]	clk	clk
8 -0.351	cnt[5]	cnt[31]	clk	clk
9 -0.295	cnt[5]	cnt[31]	clk	clk
10 -0.269	cnt[0]	cnt[26]	clk	clk
11 -0.239	cnt[0]	cnt[29]	clk	clk
12 -0.197	cnt[7]	cnt[31]	clk	clk
13 -0.149	cnt[10]	cnt[31]	clk	clk
14 -0.146	cnt[1]	cnt[30]	clk	clk
15 -0.123	cnt[0]	cnt[26]	clk	clk
16 -0.114	cnt[2]	cnt[29]	clk	clk
17 -0.093	cnt[0]	cnt[27]	clk	clk
18 -0.084	cnt[2]	cnt[30]	clk	clk
19 -0.053	cnt[10]	cnt[31]	clk	clk
20 -0.051	cnt[5]	cnt[31]	clk	clk
21 -0.016	cnt[1]	cnt[29]	clk	clk
22 -0.002	cnt[12]	cnt[31]	clk	clk

Tasks

- Report Minimum Pulse Width...
- Report False Path...
- Report Path...
- Report Exceptions...
- Report Bottlenecks...
- Report Net Timing...
- Create Slack Histograms...
- Macros
 - Report All Summaries
 - Report Top Failing Paths
 - Report All I/O Timings
 - Report All Core Timings
 - Create All Clock Histograms
- Write SDC File...
- Reset Design
- Set Operating Conditions...

Console

```

56 Info: Reading SDC File: 'hello.sdc'
57 Info: Clock uncertainty calculation is delayed until the next update_timing_netlist call.
58 update_timing_netlist
59 Info: Deriving Clock Uncertainty
60 qsta_utility::generate_top_failures_per_clock "Top failing paths" 200
61 Info: Report Timing: Found 22 setup paths (22 violated). worst case slack is -0.855
76 Info: No fmax paths to report
77 Info: No fmax paths to report
78
  
```

Появилось окно подробного отчета о пути cnt[0]/cnt[31].

Quartus II TimeQuest Timing Analyzer - D:\work_des00\tq\helloworld\helloworld - helloworld

File Edit View Netlist Constraints Reports Script Tools Window Help

Report Timing (Worst-Case Path)

Slack	From Node	To Node	Launch Clock	Latch Clock
1 -0.855	cnt[0]	cnt[31]	clk	clk

Path #1: Setup slack is -0.855 (VIOLATED)

Path Summary | Statistics | Data Path | Waveform

Total	Incr	RF	Type	Fanout	Location	Element
1 0.000	0.000					..ge ti
2 2.725	2.725	R				..k de
3 2.996	0.261		uTo	1	FF_X16_Y2_N1	cnt[0]
4 2.986	0.000	RR	CELL	2	FF_X16_Y2_N1	cnt[0]
5 3.592	0.606	RR	IC	2	LCCOMB_X16_Y2_N2	..31c
6 4.113	0.521	RR	CELL	1	LCCOMB_X16_Y2_N2	..31c
7 4.113	0.000	RR	IC	2	LCCOMB_X16_Y2_N4	..33
8 4.106	0.073	RF	CELL	1	LCCOMB_X16_Y2_N4	..33c
9 4.106	0.000	RR	IC	2	LCCOMB_X16_Y2_N4	..33c

Data Required Path

Total	Incr	RF	Type	Fanout	Location	Element
1 3.333	3.333					..ge time
2 5.962	2.629	R				..k delay
3 5.942	-0.020					..ertainty
4 5.963	0.021		uTo	1	FF_X16_Y1_N31	cnt[31]

Timing Diagram

Launch Clock Launch

Setup Relationship 3.333 ns

Latch Clock Latch

Data Arrival

Clock Delay 2.725 ns

Data Delay 4.106 ns

Slack -0.855 ns

Console

```

59 Info: Deriving Clock Uncertainty
60 qsta_utility::generate_top_failures_per_clock "Top failing paths" 200
61 Info: Report Timing: Found 22 setup paths (22 violated). worst case slack is -0.855
76 Info: No fmax paths to report
77 Info: No fmax paths to report
78 report_timing -setup -from [cnt[0]] -to [cnt[31]] -from_clock [clk] -to_clock [clk] -panel_name "Report Timing (Worst-Case Path)"
79 Info: Report Timing: Found 1 setup paths (1 violated). worst case slack is -0.855
86 1 -0.855
87
  
```

Во вкладке **Data Path** видно большое количество слоев логики между триггерами (если

судить по вкладке **Statistic** 78% всей задержки), а во вкладке **Waveform** видно как именно выглядит нарушение tsetup.

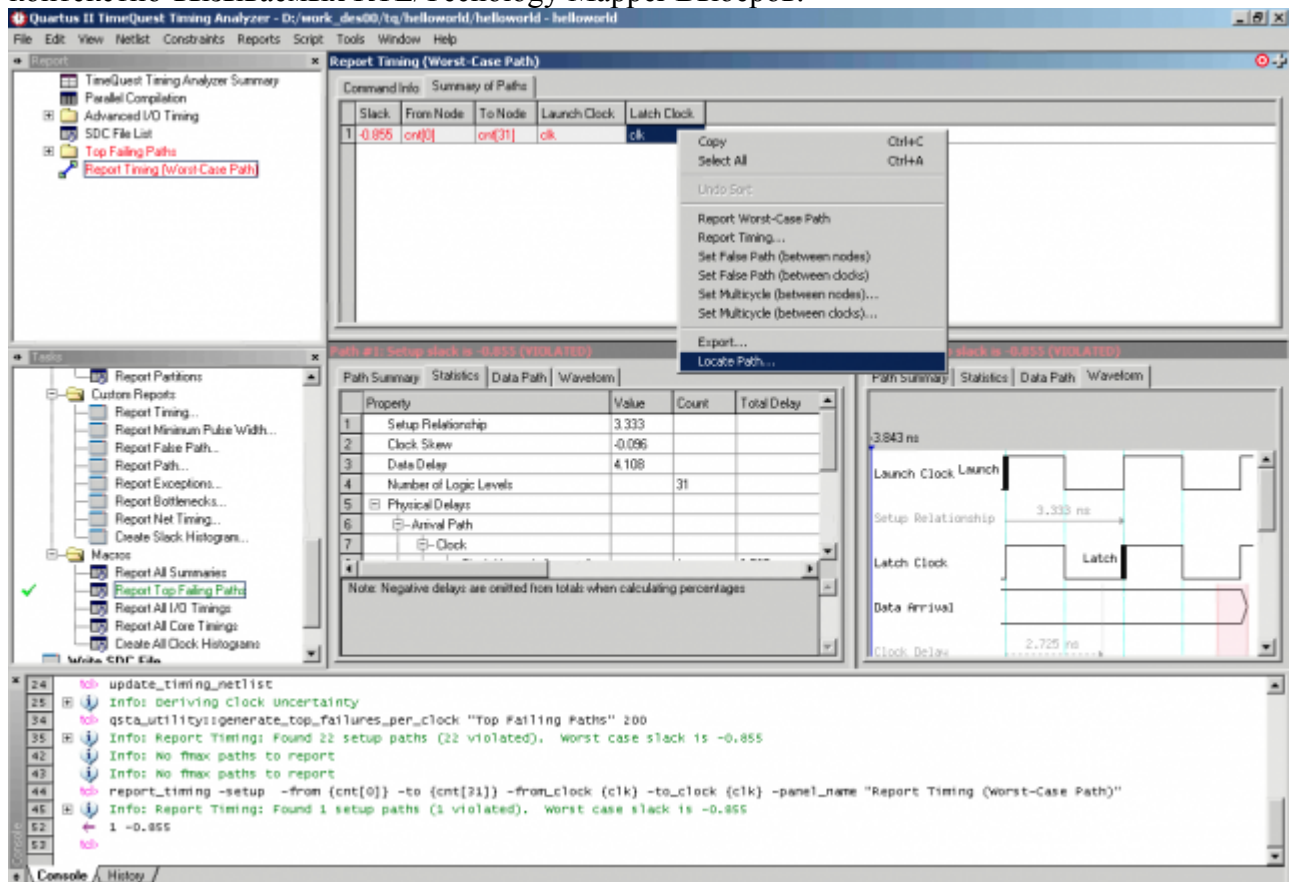
Немного подумав причину нарушения легко обосновать. Мы уперлись в задержку цепей последовательного переноса на нашем счетчике. Другими словами счетчик не успевает считать. Вариантов решения тут два: взять более быструю ПЛИС или разбить 32-х битный счетчик, например на два 16-ти битных с конвейеризированным переносом между ними.

```
module hello (input clk, output led) ;

    logic [15 : 0] cnt_low, cnt_high;
    logic          cnt_low_done;

    always_ff @(posedge clk) begin
        cnt_low      <= cnt_low + 1'b1;
        cnt_low_done <= (cnt_low == 16'hFFFE);
        if (cnt_low_done)
            cnt_high <= cnt_high + 1'b1;
        led <= cnt_high[15];
    end
endmodule
```

Вуаля, ошибок нет. Данный пример показательный, но достаточно простой. В более сложных случаях искать причину возникновения узкого места можно с помощью контекстно-вызываемых RTL/Techology Mapper Вьюеров.



По началу копаться в «мусоре» лютов и триггеров не привычно, но это дело опыта.

TimeQuest для чайников. Часть 3 (Клоки как вас много)

Продолжаем наше погружение в мир TimeQuest-a. Как вы уже поняли, эта часть посвящена проектам, в которых клоков больше чем один. Задание ограничений для таких проектов не намного сложнее, чем для рассмотренного нами ранее одночастотного проекта. Рассмотрим основные случаи многочастотных проектов.

Внимание рассматриваем **только** вопросы задания частот. Задание констрейнов ввода/вывода будем рассматривать в следующих частях.

Клоки, идущие снаружи ПЛИС

Рассмотрим простой управляемый по SPI ШИМ модулятор.

```
module spi (input clk, cs_n, sdi, sclk, output logic led) ;

//
// spi clock domain
//

logic [15 : 0] sdata;

always_ff @(posedge sclk) begin
    if (!cs_n) begin
        sdata <= (sdata << 1) | sdi;
    end
end

//
// system clock domain
//

logic      latch;
logic [2 : 0] cs_reg;
logic [15 : 0] level, cnt;

always_ff @(posedge clk) begin // simple synchronizer
    cs_reg <= (cs_reg << 1) | cs_n;
end

assign latch = ~cs_reg[2] & cs_reg[1]; // posedge

always_ff @(posedge clk) begin // spi controlled simple pwm modulator
    if (latch)
        level <= sdata;

    cnt <= cnt + 1'b1;
    led <= (cnt < level);
end

endmodule
```

Начнем составлять sdc файл. Как мы видим, в этом проекте используется два блока *clk* и *sclk*. Значит, нам нужно их описать.

```
derive_clock_uncertainty
create_clock -period 100MHz -name {clk} [get_ports {clk}]
create_clock -period 10MHz -name {sclk} [get_ports {sclk}]
```

Кроме того, как мы видим, в этом проекте данные передаются из тактового домена *sclk* в домен *clk*. Теперь внимательно посмотрим на код. Видно синхронизатор сигнала управления захвата новых данных, кроме того, видно что к моменту захвата данных в домене *clk* данные SPI будут неизменны в течении 2-х тактовых интервалов. Естественно если управляющий проц не начнет тут же новую транзакцию (даже в этом случае при отношении *clk/sclk* в 10 раз ПЛИС успеет захватить данные). Следовательно, анализировать эти пути на выполнение времянок не нужно.

Можно описать это разными способами:

1. С помощью команды *set_false_path*, указав в качестве источника и приемника, биты регистров *sdata* и *level*.

```
set_false_path -from [get_registers {sdata[*]}] -to [get_registers {level[*]}]
```

2. С помощью команды *set_false_path*, указав в качестве источника и приемника, регистры, тактируемые частотами *sclk* и *clk*.

```
set_false_path -from [get_clocks {sclk}] -to [get_clocks {clk}]
```

3. Описав группу блоков, которые асинхронны/эксклюзивны друг другу. Асинхронные клоки, в данном контексте понимаются как клоки, анализировать пути между которыми не нужно.

```
set_clock_groups -exclusive -group {clk} -group {sclk}
```

В команде используется ключ *-exclusive*, потому что эксклюзивность включает в себя свойство асинхронности клоков. У альтеры есть ключ *-asynchronous*, но он поддерживается только для совместимости со старыми версиями.

Собственно всё, мы прописали клоки проекта и указали их взаимосвязь друг с другом. Теперь квартус может оптимизировать логику доменов независимо друг от друга. И при анализе эти пути будут выброшены.

Вообще, по правде говоря, все эти 3 способа не эквивалентны друг другу. В 1ом способе не будет анализироваться конкретный путь от *sdata* до *level*. Во втором **все** пути из домена *sclk* в домен *clk*. А в третьем **все** пути из домена *sclk* в домен *clk* и **наоборот**. Какой именно метод определения false_path использовать определяется контекстом проекта.

Клоки, рожденные внутри ПЛИС

В современных проектах очень часто используется обработка сигнала или работа блоков ПЛИС на разных тактовых частотах. Это позволяет уменьшить ресурсы (меньше частота проще разводить) и энергопотребление ПЛИС. Рассмотрим два типовых случая порождения в ПЛИС дополнительного блока.

Порождение клона без PLL

Рассмотрим простой код:

```
module gen_clk (input clk_100MHz, output logic led) ;

    logic ff;

    logic [31 : 0] cnt;

    always_ff @(posedge clk_100MHz) begin // clock divider
        ff <= ~ff;
    end

    global global (ff, clk_50MHz);

    always_ff @(posedge clk_50MHz) begin
        cnt <= cnt + 1'b1;
        led <= cnt[31];
    end

endmodule
```

Видим, что с помощью делителя на триггере сделан половинный клон, который затем подан на глобальную тактовую линию и на этом клоне сделан счетчик. Подобные клоны описываются так:

```
create_generated_clock -name {clk_50MHz} -divide_by 2 -source [get_ports
{clk_100MHz}] [get_registers {ff}]
```

Здесь мы задали логическое имя клона **clk_50MHz**, источник этого клона порт ПЛИС **clk_100MHz**, преобразование частоты этого источника и объект ПЛИС на который этот клон назначен. В данном случае это триггер **ff**, на котором и производилось деление частоты. Особенное внимание нужно заострить на том факте, что источником клона указывается **не** логическое имя клона, а именно его физический источник.

Если собрать воедино всё, что мы уже узнали об sdc файлах, то для этого проекта sdc файл будет таким:

```
derive_clock_uncertainty
create_clock -period 100MHz -name {clk_100MHz} [get_ports {clk_100MHz}]

create_generated_clock -name {clk_50MHz} -divide_by 2 -source [get_ports
{clk_100MHz}] [get_registers {ff}]

set_clock_groups -exclusive -group {clk_100MHz}
set_clock_groups -exclusive -group {clk_50MHz}

set_false_path -from [get_clocks {clk_50MHz}] -to [get_ports {led}]
```

Порождение клона с помощью PLL

Рассмотрим код простого маппера, положим сигнала на входе ЦАП, который работает в режиме интерливинга входных данных.

```
module mapper (input iclk, input int idat_re, idat_im, output int odat,
output logic oval);

//
// multiply clk for output
//

pll pll(iclk, clk_x2, locked);

//
// low freq clock domain
//

int dat_re, dat_im;

always_ff @(posedge iclk) begin
    dat_re <= idat_re;
    dat_im <= idat_im;
end

//
// high freq clock domain
//

logic ff;

always_ff @(posedge clk_x2) begin
    ff <= ~ff;
end

always_ff @(posedge clk_x2) begin
    odat <= ff ? dat_re : dat_im;
    oval <= ff;
end

endmodule
```

Как мы видим, для интерливинга нужно мультиплексировать данные на выходе плис на частоте в два раза больше частоты входных символов. Первое, что нужно сделать – это описать умножение частоты. Второе – описать взаимоотношения клоков, поместив их ***iclk*** и ***clk_x2*** в **одну** клоковую группу, потому что в данном случае есть синхронная передача данных между частотами.

Описать умножение частоты на PLL можно двумя способами:

1. С помощью команды ***create_generated_clock***. В этой команде нужно прописать все параметры преобразования частоты и логическое имя клока.

```
create_generated_clock -name clk_x2 -source [get_ports {iclk}] -multiply_by 2 [get_pins {pll|altpll_component|auto_generated|pll1|clk[0]}]
```

В этом случае описание отношений частот будет такое:

```
set_clock_groups -exclusive -group {iclk clk_x2}
```

Минусом данного подхода является то, что при изменении коэффициентов PLL нужно изменять строку в sdc файле.

2. С помощью команды **derive_pll_clocks**.

```
derive_pll_clocks
```

Что происходит при выполнении этой команды можно посмотреть в консоли. А именно

Info: Deriving PLL Clocks

```
Info: create_generated_clock -source {pll|altpll_component|auto_generated|pll1|inclk[0]} -multiply_by 2 -duty_cycle 50.00 -name {pll|altpll_component|auto_generated|pll1|clk[0]} {pll|altpll_component|auto_generated|pll1|clk[0]}
```

Как мы видим, клок создан автоматически, с учетом всех преобразований. А минусом данного подхода является то, что клок получает длинное логическое имя. Описание отношений клокков соответственно будет менее элегантно:

```
set_clock_groups -exclusive -group {iclk pll|altpll_component|auto_generated|pll1|clk[0]}
```

Но есть красивый выход, использование TCL переменных (Спасибо **SM** с www.electronix.ru за помощь в этом вопросе). В этом случае описание отношений клокков будет таким:

```
set clk_x2 pll|altpll_component|auto_generated|pll1|clk[0]

set_clock_groups -exclusive -group [list $clk_x2 iclk]
```

Клоки, мультиплексируемые внутри

Иногда требуется сделать мультиплексор клокков внутри ПЛИС. В этом случае при разводке квартус должен учесть задержку тактовой частоты на мультиплексоре и обеспечить выполнение временки триггеров, которые работают на мультиплексированном клокке. Рассмотрим пример простого мультиплексора. Положим, что глитч при переключении не важен.

```
module mux (input sel, clk1, clk2, dat1, dat2, output oclk, output logic odat) ;

    assign mux_clk = sel ? clk1 : clk2;
    assign mux_dat = sel ? dat1 : dat2;
```

```
always_ff@(posedge mux_clk) begin
    odat <= mux_dat;
end
```

```
assign oclk = mux_clk;
```

```
endmodule
```

sdc файл для такого проекта выглядит очень просто

```
derive_clock_uncertainty
```

```
create_clock -period 100MHz -name {clk1} [get_ports {clk1}]
create_clock -period 100MHz -name {clk2} [get_ports {clk2}]
```

```
set_clock_groups -exclusive -group {clk1} -group {clk2}
```

Как видите командой описания отношения клоков, мы говорим, что клоки **clk1** и **clk2** представляют собой эксклюзивные группы, т.е. цепь **mux_clk** рассматривается либо нагруженная клоком **clk1**, либо клоком **clk2**. Всё, больше ничего не нужно описывать. Остальное Quartus и TimeQuest определит и сделает сам.

Клоки, мультиплексируемые снаружи

Есть системы, когда на один и тот же тактовый вход ПЛИС могут подаваться разные тактовые частоты. В этом случае надо создать несколько логических клоков на одном физическом пине.

```
create_clock -period 100MHz -name {clk_100MHz} [get_ports {clk}]
create_clock -period 50MHz -name {clk_50MHz} [get_ports {clk}] -add
create_clock -period 10MHz -name {clk_10MHz} [get_ports {clk}] -add
```

Всё, анализ времянок на выполнение будет произведен по всем этим клокам автоматически.

Итого

Как видите, алгоритм действия в многочастотных проектах везде один:

1. Описать все клоки, идущие снаружи ПЛИС (**create_clock**)
2. Описать все клоки, генерируемые внутри ПЛИС (**create_generated_clock/derive_pll_clocks**)
3. Описать их соотношения, определив группы связанных клоков (**set_clock_groups**)

Алгоритм простой и понятный как автомат Калашникова. Как и обещал, ничего сложного %).

PS. Нашел в сети заметки об особенностях команды **set_closk_groups** от Альтера Гуру

[set_clock_groups notes](#)

[Using the set_clock_groups command without hiding domain-crossing signals](#)

Рекомендую ознакомиться, особенно с первой.

TimeQuest для чайников. Часть 4 (Как много интерфейсов разных)

Вот мы и подошли к основному таинству TimeQuesta. Это таинство задания временных ограничений для интерфейсов ввода/вывода. Интерфейсов существует великое множество, но относительно методов их обработки и задания констрейнов их можно разделить на 2 основные группы: асинхронные и синхронные интерфейсы.

Под синхронностью я понимаю метод обработки интерфейса в ПЛИС. Если для обработки потока данных используется логика, тактируемая от частоты этого интерфейса, то такой интерфейс я называю синхронным. В противном случае интерфейс асинхронный.

Асинхронные я делю на истинно асинхронные (т.е. обрабатываемые полностью на комбинационной логике) и асинхронные с обработкой на системном клоке. Синхронные же в свою очередь делятся по признаку местоположения источника тактовой частоты на system-synchronous и source-synchronous интерфейсы.

Есть еще самосинхронные интерфейсы работающие с Clock Data Recovery (CDR), но их мы рассматривать не будем. Также не будем рассматривать подробно вопросы реализации интерфейсов, нас интересует только задание временных ограничений в TimeQuest.

Асинхронные интерфейсы

Истинно асинхронные интерфейсы

Иногда требуется сделать какие-нибудь простые дешифраторы команд/сигналов или делать иную комбинационную логику. В большинстве случаев можно объявить все пути как *false_path* и не париться, в любом случае задержки там будут не более 20нс и для большинства применений этого более чем достаточно.

Но есть случаи, когда нарушение времянок распространения путей чревато. Кроме того, есть особая категория пливоводов, которые любят увлекаться асинхронщиной. Очень часто этим страдают начинающие пливоводы, не читавшие библии **HDL Chip Design (c) Douglas Smith**. В таких проектах подают на тактовые входы триггеров комбинационные сигналы и т.д. Часто удивляются, почему не работает.

Работать-то оно будет, но только в том случае, если вы правильно прописали констрейны на эту асинхронщину и учли температурную зависимость длины асинхронных путей. Рассмотрим простой пример регистра защелки.

```
module async (input cs_n, we_n, input [7 : 0] idat, output [7 : 0]
odat);
    always_ff @(negedge we_n) begin
        odat <= ~cs_n ? idat : odat;
    end
endmodule
```

sdc файл для такого проекта, на сыклоне 3, может быть таким

```
create_clock -period 10MHz -name {write} [get_ports {we_n}]
```

```
set_max_delay -from [get_ports {cs_n}] -to [get_pins {odat[*]~reg0\ena}] 5.5ns
```

```
set_min_delay -from [get_ports {cs_n}] -to [get_pins {odat[*]~reg0\ena}] 4.5ns
```

```
set_max_delay -from [get_ports {idat[*]}] -to [get_pins {odat[*]~reg0\d}] 5.5ns
```

```
set_min_delay -from [get_ports {idat[*]}] -to [get_pins {odat[*]~reg0\d}] 4.5ns
```

```
set_max_delay -from [get_pins {odat[*]~reg0\q}] -to [get_ports {odat[*]}] 5.5ns
```

```
set_min_delay -from [get_pins {odat[*]~reg0\q}] -to [get_ports {odat[*]}] 4.5ns
```

Ага, скажут многие, а клок-то все-таки есть. Да есть, но нужен он для того, чтобы TimeQuest не кричал о нарушении **Minimal Pulse Width** сигнала приходящего на тактовые порты триггеров.

Как вы видите, для данной схемы пришлось прописывать все основные задержки по путям, при этом применять их надо к конкретным пинам триггера ПЛИС. При смене семейства ПЛИС потребуются правка этих пинов в sdc файле.

Собственно все, собираем проект, запускаем анализ и в итоге видим:

	Slack	From Node	To Node	Launch Clock	Latch Clock
1	-0.252	idat[5]	odat[5]~reg0	n/a	write
2	-0.103	idat[2]	odat[2]~reg0	n/a	write
3	-0.085	idat[3]	odat[3]~reg0	n/a	write
4	-0.050	idat[7]	odat[7]~reg0	n/a	write
5	0.001	idat[4]	odat[4]~reg0	n/a	write
6	0.282	cs_n	odat[5]~reg0	n/a	write
7	0.315	cs_n	odat[0]~reg0	n/a	write
8	0.315	cs_n	odat[1]~reg0	n/a	write
9	0.347	cs_n	odat[6]~reg0	n/a	write
10	0.347	cs_n	odat[7]~reg0	n/a	write
11	0.359	cs_n	odat[2]~reg0	n/a	write
12	0.359	cs_n	odat[3]~reg0	n/a	write
13	0.359	cs_n	odat[4]~reg0	n/a	write

Квартусу не хватает ума, что бы с помощью LUT'ов выровнять задержки по некоторым путям, делать это придется в рукопашную, вставляя lcell буферы. Или расширить диапазон требуемой задержки. Другого пути я не вижу.

Видя такую особенность квартуса, вспоминается старая реклама "Вы все еще балуетесь асинхронщиной? Тогда мы идем к вам...". На самом деле никуда мы не идем, решение о том, что использовать, а что нет, в вашем проекте лежит целиком на вас.

Асинхронные интерфейсы с обработкой на системном клоке

В качестве яркого примера асинхронного интерфейса можно взять всем известный, самый обычный UART.

Приемный сигнал UART_TX -> FPGA_RX нарезают системной частотой, которая должна быть выше символьной частоты UART и обрабатывают, используя детекторы перехода сигнала из состояния в состояние.

Констрейны на такие интерфейсы это уже знакомые нам `set_false_path`:

```
set_false_path -from [get_ports {uart_rx}] -to [get_clocks {sys_clk}]
```

Так мы задаем путь, который не надо анализировать от порта `uart_rx` до триггеров тактируемых клоком `sys_clk`:

```
set_false_path -from [get_ports {uart_rx}] -to [all_clocks]
```

А так путь от порта `uart_rx` до триггеров, тактируемых от **любого** из клоков в системе. С выходным сигналом поступаем точно так же:

```
set_false_path -from [all_clocks] -to [get_ports {uart_tx}]
```

Стоит отметить, что в данном случае (UART) можно даже не контролировать использование триггера в I/O буфере ПЛИС.

Рассмотрим теперь более сложный пример, возьмем простой SPI мастер. Положим, что данные захватываются слейвом по фронту тактовой частоты.

```
module spi (input clk, start, input [7 : 0] data, output logic busy,  
            sclk, sdi, cs_n) ;
```

```
    logic      ff ;  
    logic [3 : 0] cnt ;  
    logic      done ;  
    logic [7 : 0] buffer ;
```

```
    assign ff = cnt[0] ;  
    assign done = &cnt ;
```

```
    always_ff @(posedge clk) begin // simple FSM  
        if (~busy) begin  
            if (start) begin  
                busy  <= 1'b1 ;  
                cnt   <= 0 ;  
                buffer <= data ;  
            end  
        end  
    end
```

```

else begin
    cnt <= cnt + 1'b1;
    if (ff) begin
        buffer <= (buffer << 1);
    end
    if (done) begin
        busy <= 1'b0;
    end
end
end
end

always_ff @(posedge clk) begin // io registers
    sclk <= ff;
    sdi <= buffer[7];
    cs_n <= ~busy;
end

endmodule

```

По идее, этот интерфейс синхронный, т.е. вместе с данными передается сигнал тактовой частоты `sclk`. Но, как мы видим из кода, временные отношения между сигналами заданы последовательностью состояний конечного автомата. И единственное, что нужно для их четкого выполнения, убедиться, что выходные триггеры были размещены в I/O ячейках. Как вы уже, наверное, догадались, констрейны для данного интерфейса будут:

```
set_false_path -from [get_clocks {clk}] -to [get_ports {sclk sdi cs_n}]
```

Синхронные интерфейсы

Для разговора о задании констрейнов для синхронных интерфейсов нужно въехать в основы временного анализа. Здесь мой читатель придется уже поработать вам, т.к. пересказывать альтеровские документы у меня нет никакого желания. Да и документы эти небольшие и не сложные. Советую читать вот в таком порядке:

1. `Clock_Setup_and_Hold_Slack_Explained.doc` - что бы понять, что такое setup/hold и как они анализируются.
2. `an433.pdf` - `Constraining and Analyzing Source-Synchronous Interfaces` – чтобы еще раз прочитать про задание констрейнов для синхронных интерфейсов и заодно проверить меня %).
3. `mn1_timequest_cookbook.pdf` - в подарок краткий справочник о TimeQuest, в нем в краткой форме содержится все то, что мы уже усвоили + кое-что вкусненькое.

Без изучения этих документов дальнейшие примеры будут мало полезны. Положим, что мы ознакомились с содержимым файла `Clock_Setup_and_Hold_Slack_Explained.doc`.

Подобьем нужную нам информацию, перед тем как идти дальше.

1. Синхронные интерфейсы рассматривают передачу данных от регистров, тактируемых частотой источника, до регистров, тактируемых частотой приемника.
2. TimeQuest не телепат, задача описания этих клоков и их временного соотношения целиком ваша.

3. Для выходных интерфейсов надо помнить, что у триггера есть два важных параметра ***Tsetup(tsu)*** – время предустановки данных (время, в течение которого данные должны быть неизменны до фронта тактовой частоты) и ***Thold(th)*** - время удержания данных (время, в течение которого данные должны быть неизменны после фронта тактовой частоты). Если эти времена нарушаются, то возможны сбои в работе логики, эти сбои называются метастабильностью.
4. Для входных интерфейсов надо помнить что, в зависимости от реализации источника (регистр/АЦП/память/и т.д.) могут быть следующие важные параметры. Для регистров это ***Tclock-to-out(tco)*** - время появления данных на выходе после фронта тактовой частоты. Для памяти это ***Tacces*** - время появления данных на выходе после фронта тактовой частоты, ***Thold*** - время удержания данных на выходе после фронта тактовой частоты.
5. TimeQuest при анализе синхронных интерфейсов ставит клоки, относительно друг друга, в то положение, которое вы ему указали. Если вы ничего не указывали, то он ставит клоки в положение фронт в фронт.

Если вы не указали любой из клоков, то вините себя, т.к. TimeQuest может выдать все что угодно.

Синхронные интерфейсы разделяются на два вида по месту происхождения источника тактирования интерфейсного устройства:

1. System Synchronus - это интерфейсы, в которых тактовая частота интерфейса идет непосредственно с ПЛИС на интерфейсное устройство. К таким интерфейсам можно отнести АЦП/ЦАП, память, шину в режиме master и т.д.
2. Source Synchronus - это интерфейсы, в которых тактовая частота идет от интерфейсного устройства к ПЛИС. Или от отдельного генератора к ПЛИС и интерфейсному устройству, через всякие разветвители. К таким интерфейсам можно отнести шину процессора, к которому ПЛИС подключена как slave, АЦП/ЦАП и т.д.

System-Synchronus Output

Рассмотрим вывод пины на синхронный параллельный ЦАП.

```
module dac (input iclk, output oclk, output logic [7 : 0] data);

    logic [7 : 0] cnt;

    always_ff @(posedge iclk) begin
        cnt <= cnt + 1'b1;
        data <= cnt;
    end

    assign oclk = iclk;

endmodule
```

Положим параметры ЦАП ***tsu/th*** = 5ns/5ns, частота 10МГц, все настройки стоят по *default*. sdc файл для данного проекта будет таким.

```
set_time_format -unit ns -decimal_places 3
```

```
derive_clock_uncertainty
```

```
create_clock -period 10MHz -name {iclk} [get_ports {iclk}]
```

```
create_generated_clock -name {oclk} -source [get_ports {iclk}] [get_ports {oclk}]
```

```
set_output_delay -clock [get_clocks {oclk}] -max 5.0 [get_ports {data[*]}]
```

```
set_output_delay -clock [get_clocks {oclk}] -min -5.0 [get_ports {data[*]}]
```

Поясним, что означают незнакомые/непонятные для нас строки:

```
set_time_format -unit ns -decimal_places 3
```

Мы задаем единицы измерения времени, чтобы не писать их по месту.

```
create_generated_clock -name {oclk} -source [get_ports {iclk}] [get_ports {oclk}]
```

Это описание блока, на котором работает приемник нашего интерфейса (ЦАП). Как видно из кода, никаких преобразований с блоком не было, поэтому он назначен один в один.

```
set_output_delay -clock [get_clocks {oclk}] -max 5.0 [get_ports {data[*]}]
```

```
set_output_delay -clock [get_clocks {oclk}] -min -5.0 [get_ports {data[*]}]
```

Это описание констрейна связанное с *tsu* и *th* ЦАПа соответственно. По идее эти задержки должны задаваться как

*Output maximum delay value = maximum trace delay for data + tSU of external register
- minimum trace delay for clock*

*Output minimum delay = minimum trace delay for data - tH of external register –
maximum trace delay for clock*

Но в данном случае мы ими пренебрегаем. Положим, что задержки выровнены на плате. Всё, мы задали констрейны для нашего проекта. Собираем, запускаем анализ и видим в логах квартуса:

Critical Warning: Timing requirements not met

Казалось бы всего 10МГц, как же так. Давайте разбираться подробно. Запускаем TimeQuest, выполняем анализ и видим результат:

The screenshot displays the Quartus II Timing Analyzer interface. The main window shows a table titled 'Hold Clock' with the following data:

Slack	From Node	To Node	Launch Clock	Latch Clock
1 3.437	data[0]neg[0] data[0]	data[0]	clock	clock
2 3.433	data[3]neg[0] data[3]	data[3]	clock	clock
2 3.431	data[5]neg[0] data[5]	data[5]	clock	clock
4 3.454	data[2]neg[0] data[2]	data[2]	clock	clock
5 3.260	data[7]neg[0] data[7]	data[7]	clock	clock

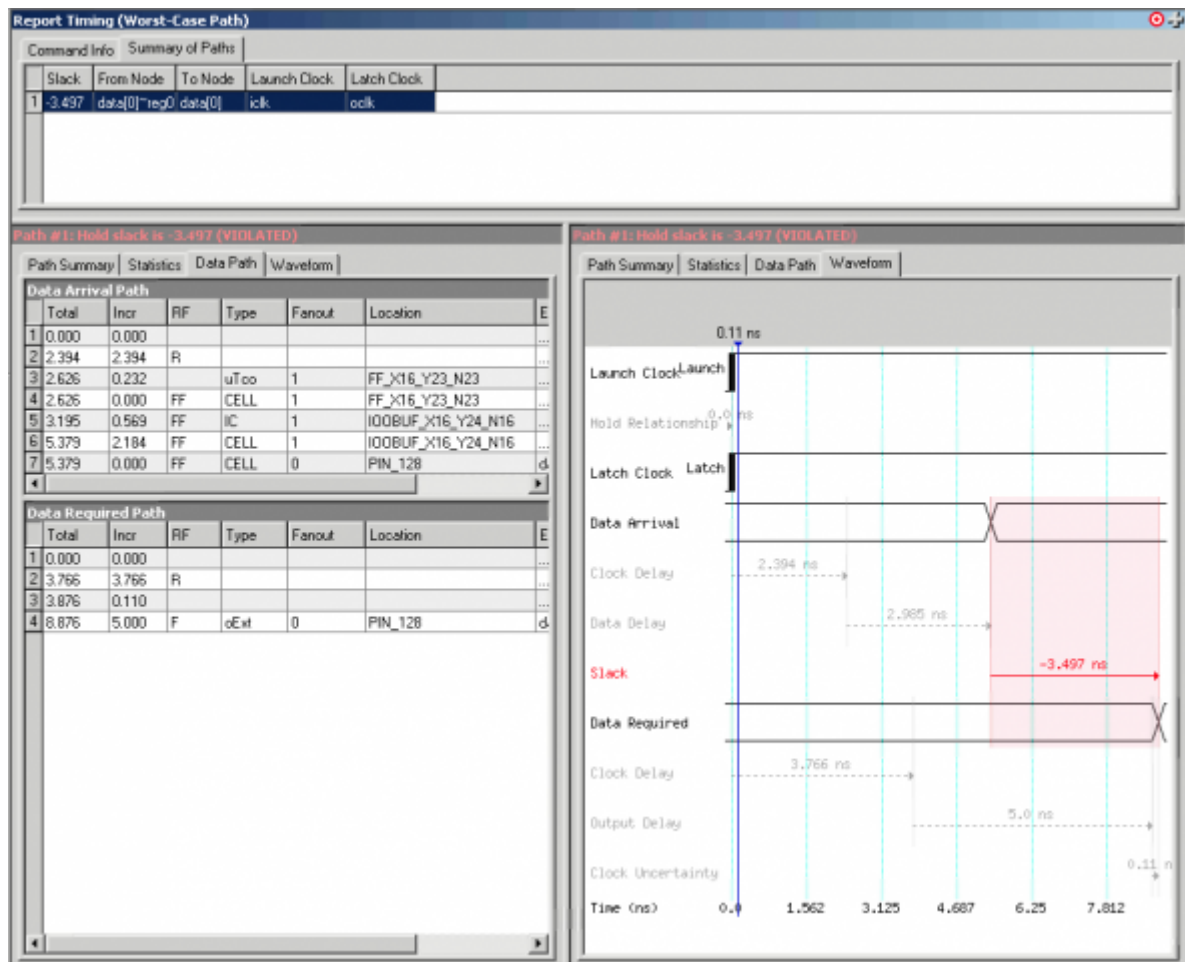
The 'Timing' report tree on the left includes sections for 'Report Clocks', 'Report Clock Transfers', 'Report Unconstrained Paths', 'Report SDC', 'Report Ignored Constraints', 'Check Timing', 'Report Partitions', 'Custom Reports', and 'Macros'. The 'Macros' section is expanded, showing 'Report All Summaries', 'Report Top Failing Paths', 'Report All I/O Timings', 'Report All Core Timings', and 'Create All Clock Histograms'. The 'Report Top Failing Paths' macro is selected.

The 'Console' window at the bottom shows the following log messages:

```

170: tcl: read_sdc
171: Info: Reading SDC File: 'dac.sdc'
172: Info: Clock uncertainty calculation is delayed until the next update_timing_netlist call.
173: tcl: update_timing_netlist
174: Info: Deriving Clock Uncertainty
175: qsta_utility::generate_top_failures_per_clock "Top Failing Paths" 200
176: Info: Report Timings: Found 5 hold paths (5 violations). Worst case slack is -3.497
177: Info: No max paths to report
178: Info: No max paths to report
179: tcl:
  
```

И-да, действительно не укладываемся во времянку. Давайте выясним почему. С помощью **Report Worst Case Path** смотрим, что же происходит в интерфейсе.



По началу кажется, что на вейвформе изображен сплошной бред. Но не будем торопиться.

Смотрим внимательно, видно **Launch Clock = iclk** и **Latch Clock = oclk**. Они совпадают. Неужели TimeQuest забыл про задержку? А вот и нет, просто оба эти клона идут от входного порта ПЛИС **iclck** и временной анализ начинается с этой точки.

Смотрим далее. Видим **Clock Delay = 2.394нс** и **Data Delay = 2.985нс**. **Clock Delay** – это задержка от порта ПЛИС **iclck**, до тактового входа триггера **data[0]**. А **Data Delay** – это задержка от выхода триггера **data[0]** до порта плис **data[0]**. Сумма этих величин дает **Data Arrival**, т.е. задержку прибытия данных на порт плис.

Теперь смотрим, что происходит с клоком **oclk**. Чуть ниже видим **Clock Delay = 3.766нс**. Это и есть задержка клона **oclk** относительно клона **iclck**. Прибавляем к этому времени требуемое нам **th** (мы же не проходим по холду) и получаем **Data Required**. Как мы видим, а ведь действительно условие по **th** не выполняется.

А может быть, TimeQuest врет, проверим задержку между портами **iclck** и **oclk**. Выполним task **Report Path...**

Report Path

Targets

From: [get_ports {iclk}] ...

Through: ...

To: [get_ports {oclk}] ...

Paths

Report number of paths: 1

Output

☒ Report panel name: Report Path

☐ File name: ...

File options

☒ Overwrite ☐ Append

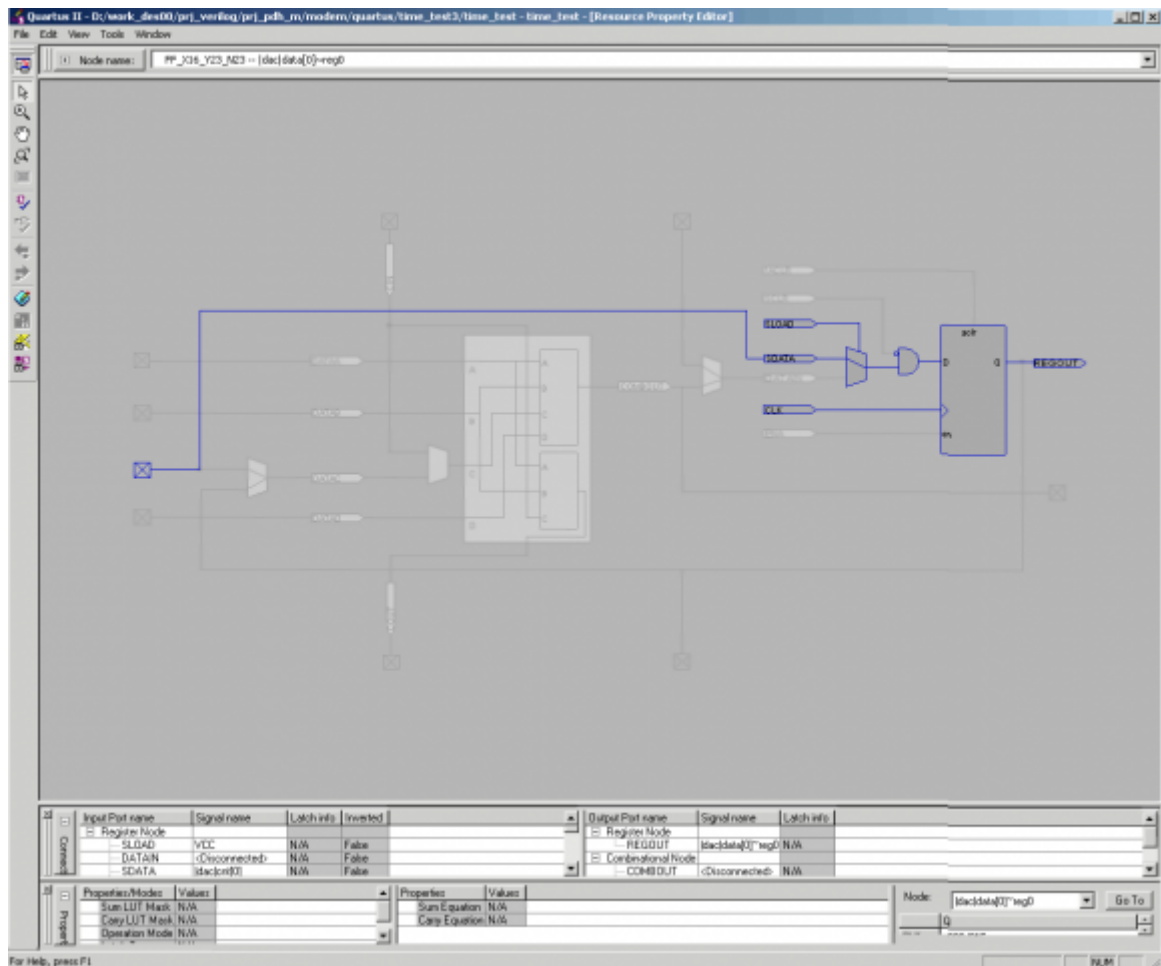
Open

☐ Console

Tcl command: report_path -from [get_ports {iclk}] -to [get_ports {oclk}] -npaths 1 -p

Report Path Close Help

Действительно, не врет.



Уп-с, а триггер-то не в IO буфере вообще, а без этого триггера циклон 3 не умеет использовать задержку в I/O буфере. Назначаем на *data[*]* **Fast Output Register**. Проверяем и ...

Он сделал все, что мог, поставил триггер в I/O буфер, и даже использовал задержку (*Output Pin Delay == 1*). Но, "ну не шмогла я не шмогла" (с) Старый анекдот.

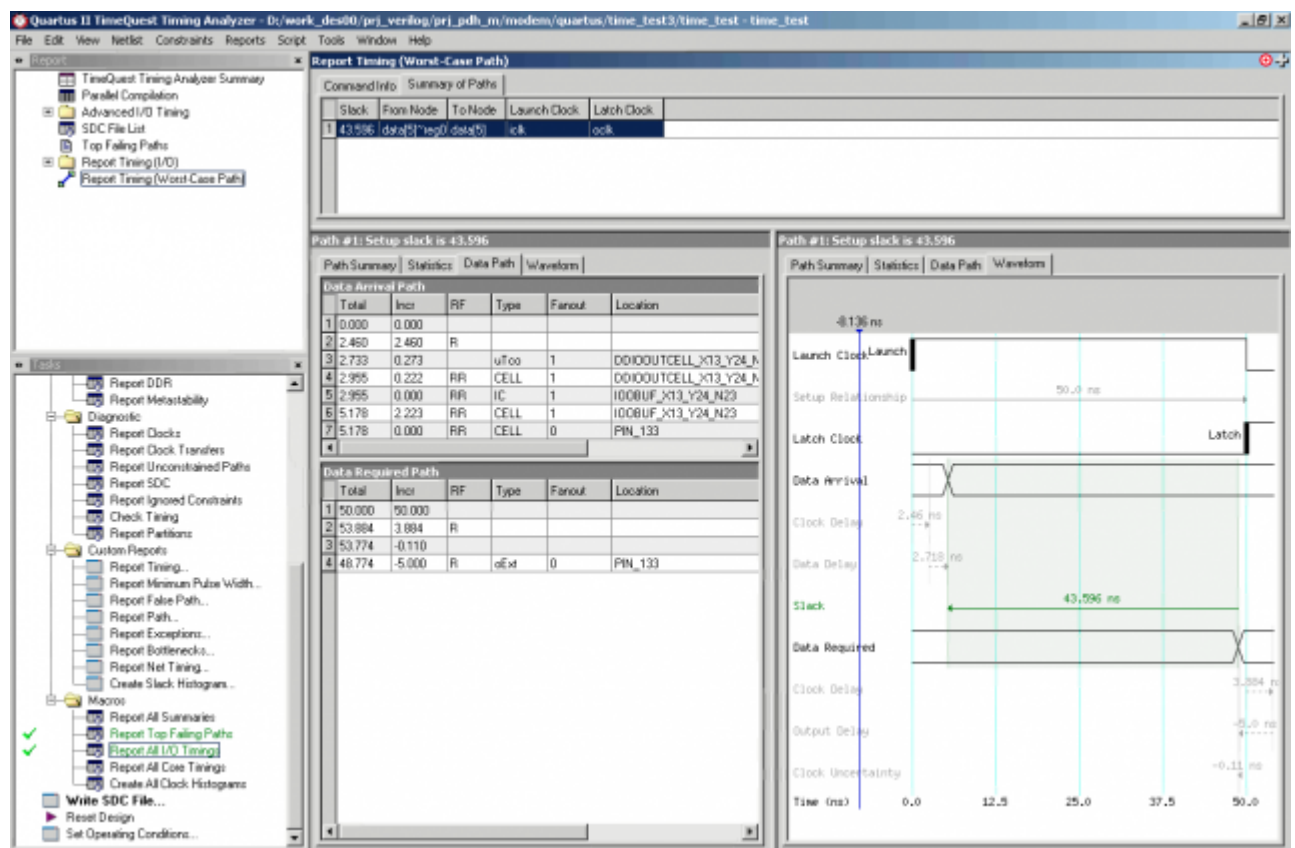
И тут мы вспоминаем, что, подав инверсный клок, мы поставим фронт как раз в середину данных, тогда *tsu/th* должны вылезти как бы автоматом. Пишем в коде

```
assign oclk = ~iclk;
```

В sdc файле

```
create_generated_clock -name {oclk} -invert -source [get_ports {iclk}] [get_ports {oclk}]
```

Вуаля, ошибок нет. И еще больше 40нс в запасе.

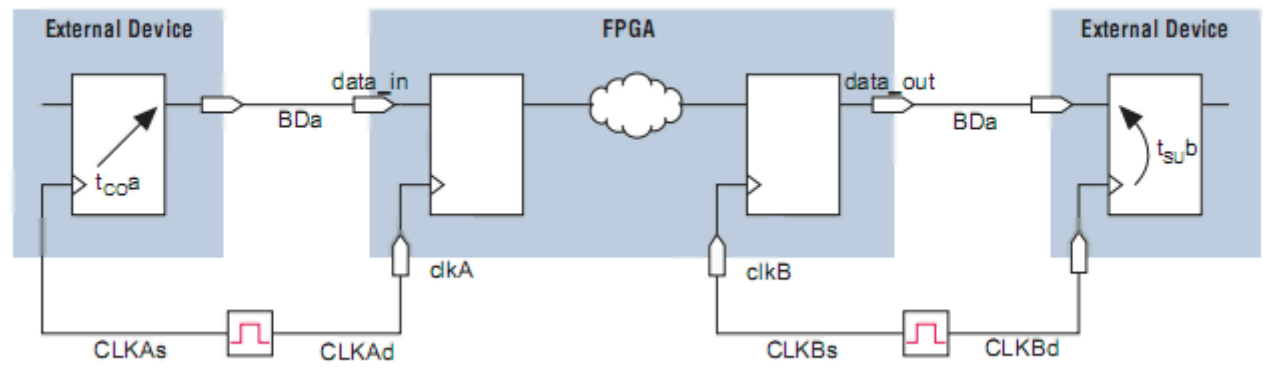


Вот в принципе и все про данный вид интерфейсов. От себя хочу добавить, что метод инверсии клона не является универсальным. И использовать его везде не следует.

Source-Synchronous Output

Как уже обсуждалось выше, в подобных системах тактовая частота на ПЛИС идет с приемника данных или с отдельного генератора, частота которого идет на ПЛИС и периферийное устройство. Данный метод тактирования используется в случае, когда требуется минимизировать джиттер тактового сигнала, это особенно важно для АЦП/ЦАП. Пример построения такой системы:

Figure 1–9. Chip-to-Chip Design



Как видно, в этом случае источник тактовой частоты для приемника интерфейсов вообще не присутствует в ПЛИС. Как же его описать? Для описания таких ситуаций в TimeQuest есть возможность задать так называемый виртуальный клок. Т.е. клок, у которого отсутствует физический источник.

Рассмотрим опять вывод пилы на ЦАП, на сей раз с внешним тактированием.

```
module dac (input clk, output logic [7 : 0] data);  
  
    logic [7 : 0] cnt;  
  
    always_ff @(posedge clk) begin  
        cnt <= cnt + 1'b1;  
        data <= cnt;  
    end  
  
endmodule
```

Соответствующий этому проекту sdc файл

```
set_time_format -unit ns -decimal_places 3  
  
derive_clock_uncertainty  
  
create_clock -period 10MHz -name {clk} [get_ports {clk}]  
  
create_clock -period 10MHz -name {virt_clk}  
  
set_clock_groups -exclusive -group {clk virt_clk}  
  
# увеличение уменьшает запас по tsu  
set CLK_bs_delay_max 0.2  
set DATA_delay_max 0.5  
# увеличение увеличивает запас по tsu  
set CLK_bd_delay_min 1.0  
  
# увеличение увеличивает запас по th  
set CLK_bs_delay_min 0.2
```



```

set DATA_delay_min    0.5
#увеличение уменьшает запас по th
set CLK_bd_delay_max    1.0

set tSU                5.0
set tH                 5.0

set_output_delay -clock [get_clocks {virt_clk}] \
    -max [expr $CLK_bs_delay_max + $tSU + $DATA_delay_max -
$CLK_bd_delay_min] [get_ports {data[*]}]

set_output_delay -clock [get_clocks {virt_clk}] \
    -min [expr $CLK_bs_delay_min - $tH + $DATA_delay_min - $CLK_bd_delay_max]
[get_ports {data[*]}]

```

Первый взгляд на sdc файл вызывает в голове мысли "А ну нафиг такие интерфейсы, давайте тактировать от ПЛИС". Но не надо торопиться, здесь все логично и просто. Начнем разбирать те строки, которые нам еще не знакомы.

ЦАП тактируется от того же генератора что и ПЛИС, т.е. в плис этот клок физически не существует, но относительно ПЛИС он есть. Эта строка

```
create_clock -period 10MHz -name {virt_clk}
```

и описывает тот самый клок. Теперь нужно задать констрейны на *tsu/th*

```

set_output_delay -clock [get_clocks {virt_clk}] \
    -max [expr $CLK_bs_delay_max + $tSU + $DATA_delay_max -
$CLK_bd_delay_min] [get_ports {data[*]}]

set_output_delay -clock [get_clocks {virt_clk}] \
    -min [expr $CLK_bs_delay_min - $tH + $DATA_delay_min - $CLK_bd_delay_max]
[get_ports {data[*]}]

```

Внимательно приглядевшись, мы видим общее с примером system-synchronous output рассмотренным ранее. Это указание времен tsu/th в задержках. Но также видим отличия. Заключаются они вот в чем:

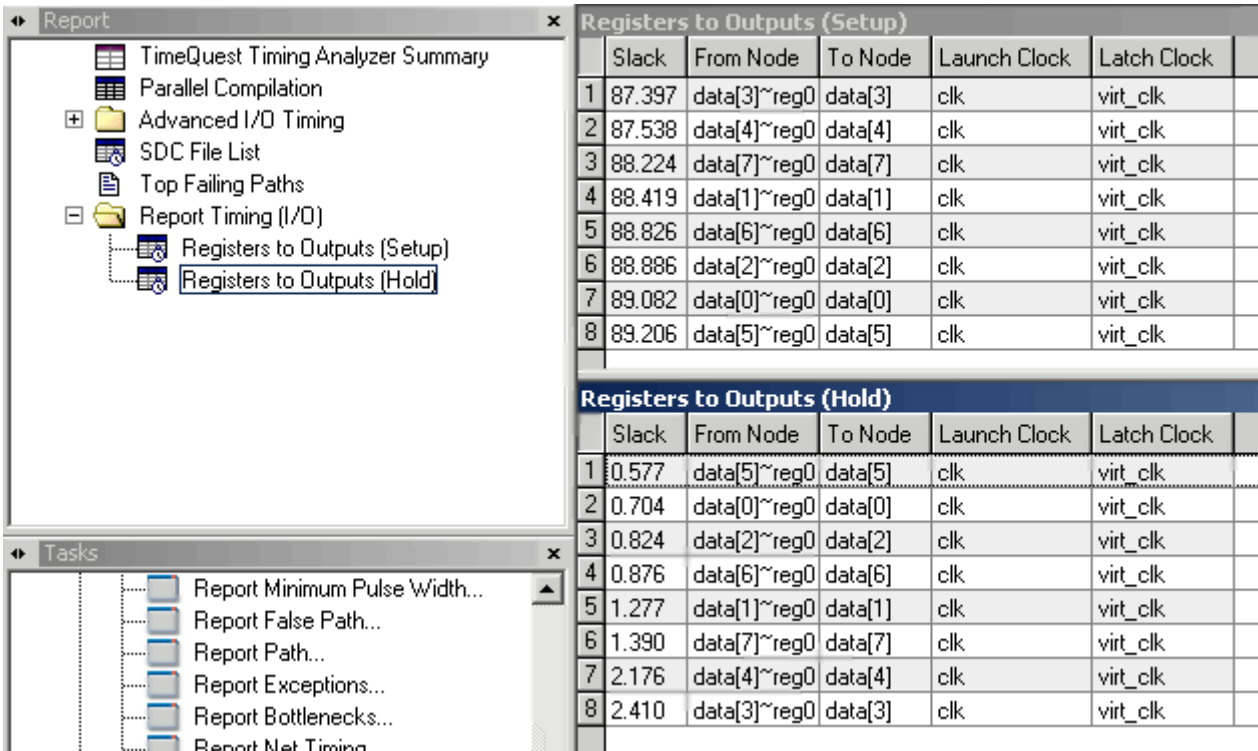
1. Клоки тактирования ЦАП и ПЛИС идут разными путями на плате, то для временного анализа надо знать их местоположения друг относительно друга. А это мы можем извлечь только из конструктива печатной платы. **Поэтому эти задержки нужно измерить и учесть в констрейне.**
2. Данные тоже проходят определенный путь по плате и этот путь отличен от пути блока ЦАП, поэтому **задержку данных тоже нужно обязательно учесть.**
3. Чтобы не считать все задержки в уме (а если вы их будете делать это часто, то это лишний повод для ошибки), мы используем возможность TimeQuest считать выражения самостоятельно. Делается это с помощью TCL переменных и команды расчета выражений *[expr]*

В приведенном выше sdc файле видно, что используются максимальные и минимальные значения задержек. Сделано это потому, что задать задержку на ПП можно с

определенной точностью, а для анализа нас интересует наихудший случай. Потому-то при заданиях констрейнов и используются разные значения задержек.

В скрипте, как вы видите, значения минимальных и максимальных задержек одинаковые, потому что на изложение теории это не влияет (формулы написаны верные) и это скрытый намек для вас поиграть с этими задержками в TimeQuest и посмотреть результат

Собираем проект, запускаем временной анализ. Смотрим:

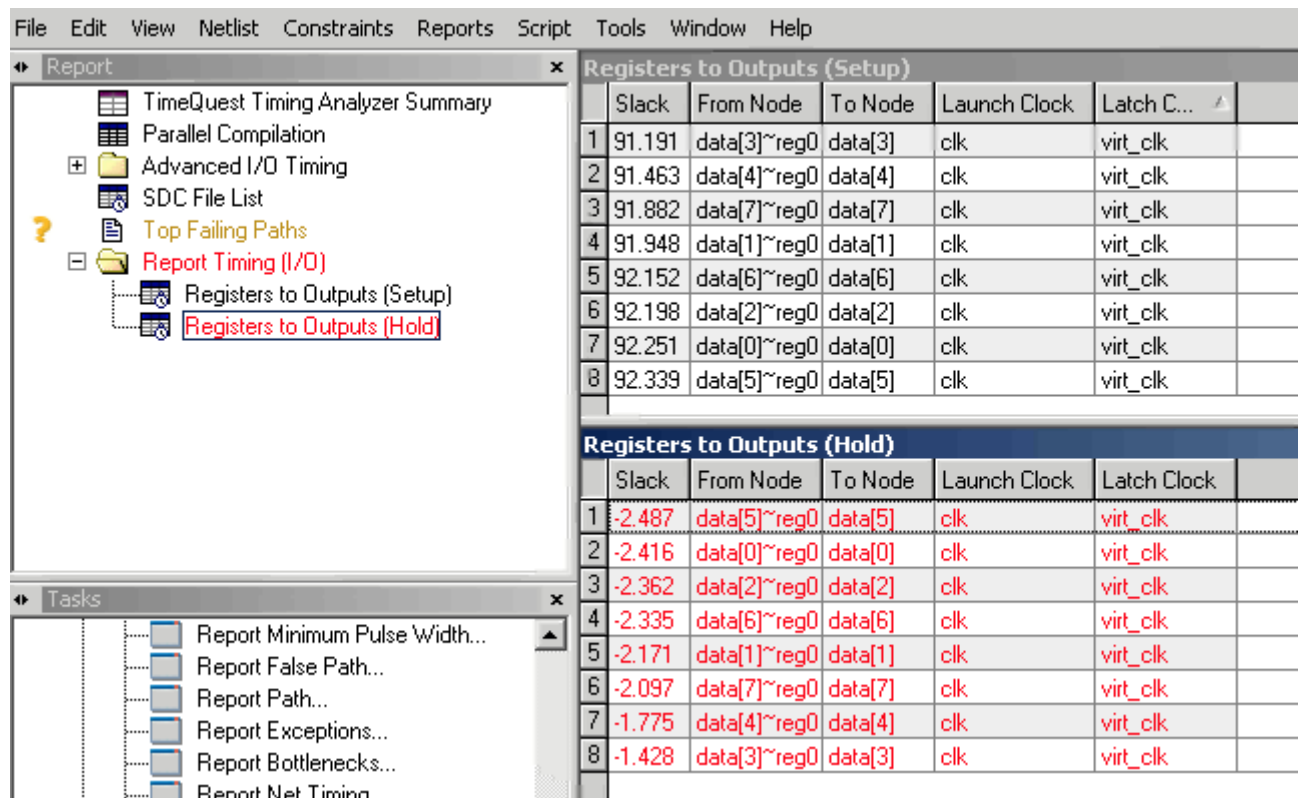


The screenshot shows the TimeQuest Timing Analyzer interface. The 'Report' window is open, displaying a tree view on the left with 'Registers to Outputs (Setup)' and 'Registers to Outputs (Hold)' selected. The 'Tasks' window is also open, showing a list of tasks. The main area displays two tables: 'Registers to Outputs (Setup)' and 'Registers to Outputs (Hold)'.

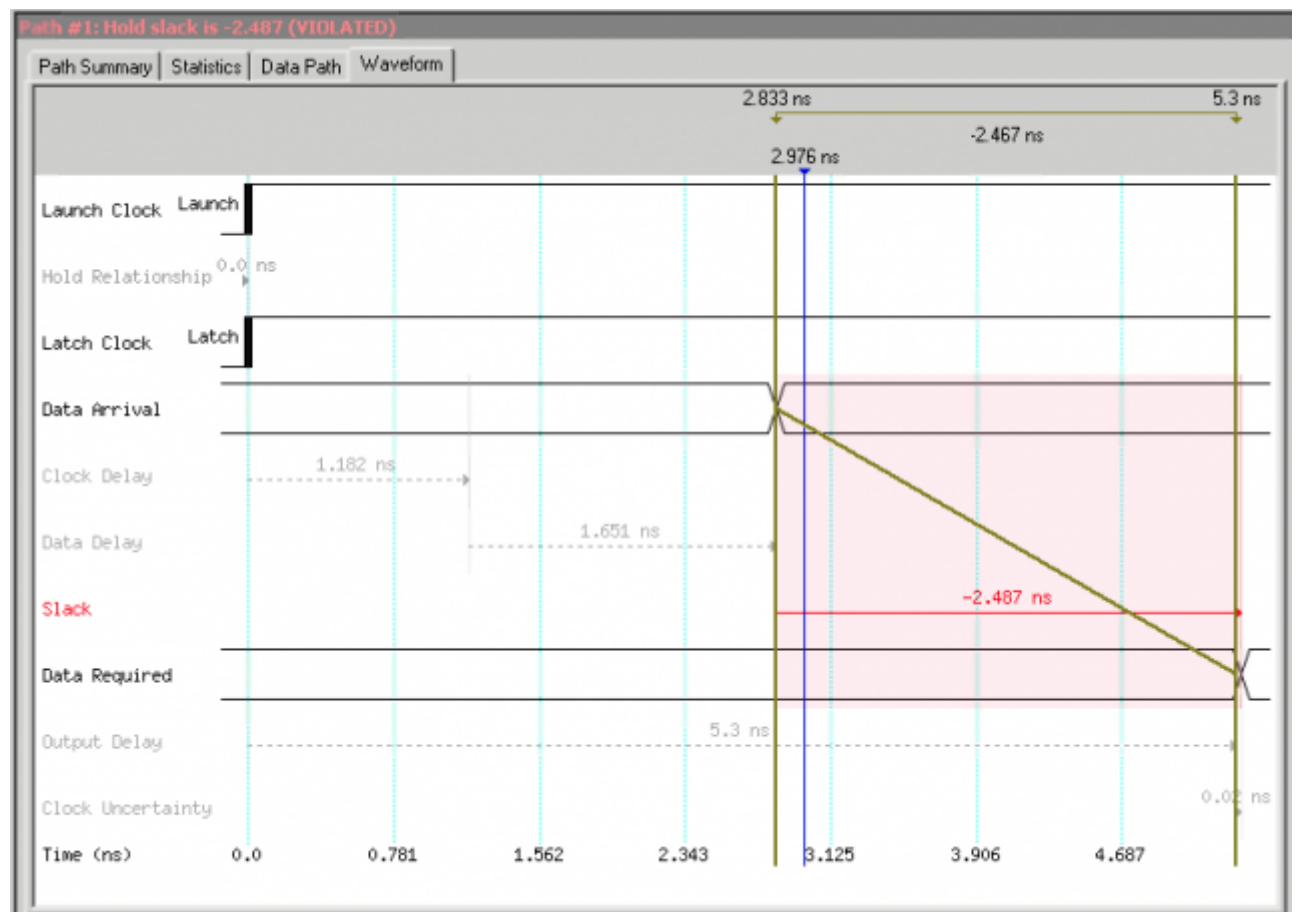
	Slack	From Node	To Node	Launch Clock	Latch Clock
1	87.397	data[3]~reg0	data[3]	clk	virt_clk
2	87.538	data[4]~reg0	data[4]	clk	virt_clk
3	88.224	data[7]~reg0	data[7]	clk	virt_clk
4	88.419	data[1]~reg0	data[1]	clk	virt_clk
5	88.826	data[6]~reg0	data[6]	clk	virt_clk
6	88.886	data[2]~reg0	data[2]	clk	virt_clk
7	89.082	data[0]~reg0	data[0]	clk	virt_clk
8	89.206	data[5]~reg0	data[5]	clk	virt_clk

	Slack	From Node	To Node	Launch Clock	Latch Clock
1	0.577	data[5]~reg0	data[5]	clk	virt_clk
2	0.704	data[0]~reg0	data[0]	clk	virt_clk
3	0.824	data[2]~reg0	data[2]	clk	virt_clk
4	0.876	data[6]~reg0	data[6]	clk	virt_clk
5	1.277	data[1]~reg0	data[1]	clk	virt_clk
6	1.390	data[7]~reg0	data[7]	clk	virt_clk
7	2.176	data[4]~reg0	data[4]	clk	virt_clk
8	2.410	data[3]~reg0	data[3]	clk	virt_clk

видим, что мы уложились в констрейны, но посмотрите, какой большой запас есть по *tsu*, и какой маленький запас по *th*. Работать с таким запасом крайне рискованно, даже на 10МГц. Кроме того, в режиме работы *fast*, видно что у нас вылезают слаки по холду:



если посмотреть поподробнее,



то видно, что нам не хватает задержки по данным. А у квартуса не хватает ума, чтобы набрать эту задержку, например на LUT'ах. В данном примере Fast Output Register не

используется, можно попробовать задержать данные на задержках в I/O буферах. Но в таком случае более эффективна другая техника. С помощью PLL подвинуть клок ПЛИС.

Генерим PLL, работающую один в один и сдвигающую клок по фазе на 90 градусов.

```
module dacpll (input clk, output logic [7 : 0] data);

    pll_no_delay
    pll_no_delay
    (
        .areset (1'b0),
        .inclk0 (clk),
        .c0      (used_clk),
        .locked ( )
    );

    logic [7 : 0] cnt;

    always_ff@(posedge used_clk) begin
        cnt <= cnt + 1'b1;
        data <= cnt;
    end

endmodule
```

Вуаля, запас по холду вырос до 23.5нс.

Registers to Outputs (Setup)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	64.201	data[4]~reg0	data[4]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
2	65.550	data[1]~reg0	data[1]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
3	65.558	data[3]~reg0	data[3]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
4	65.617	data[0]~reg0	data[0]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
5	65.617	data[7]~reg0	data[7]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
6	65.628	data[5]~reg0	data[5]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
7	65.629	data[6]~reg0	data[6]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
8	65.639	data[2]~reg0	data[2]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
Registers to Outputs (Hold)						
	Slack	From Node	To No...	Launch Clock	Latch Clock	
1	23.559	data[2]~reg0	data[2]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
2	23.569	data[6]~reg0	data[6]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
3	23.570	data[5]~reg0	data[5]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
4	23.571	data[0]~reg0	data[0]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
5	23.571	data[7]~reg0	data[7]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
6	23.654	data[3]~reg0	data[3]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
7	23.662	data[1]~reg0	data[1]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	
8	25.101	data[4]~reg0	data[4]	pll_no_delay\altpll_component\auto_generated\pll1 clk[0]	virt_clk	

а все потому, что мы подвинули клок на 25нс (1/4 периода частоты 10МГц).

Как вы понимаете, случай тактирования от одного генератора, ничем не отличается от случая тактирования от приемника данных. Просто в одном случае у нас должны быть учтены длины трас до обоих приемников тактового сигнала, а в другом случае только от приемника данных до ПЛИС.

System-Synchronus Input

Рассмотрим теперь тактирование АЦП от ПЛИС. В качестве примера возьмем параллельный АЦП AD9215. Положим что частота тактирования этого АЦП 100 МГц.

```
module adc (input clk, input [9 : 0] adc_dat, output adc_clk, output
logic [9 : 0] data);

logic [9 : 0] adc_io_reg;
logic [9 : 0] adc_reg;

always_ff @(posedge clk) begin
    {adc_reg, adc_io_reg} <= {adc_io_reg, adc_dat};
end

assign adc_clk = clk;

always_ff @(posedge clk) begin
    data <= adc_reg;
end

endmodule
```

Начальный (как вы уже, наверное, поняли, в процессе ковыряния проекта, часто требуется рихтовка sdc файла) sdc файл такой:

```
set_time_format -unit ns -decimal_places 3

derive_clock_uncertainty

create_clock -period 100MHz -name {clk} [get_ports {clk}]

create_generated_clock -name {adc_clk} -source [get_ports {clk}] [get_ports {adc_clk}]

set ADC_CLK_delay_max [expr 30.0*0.007]
set ADC_CLK_delay_min [expr 30.0*0.007]
set ADC_DATA_delay_max [expr 20.0*0.007]
set ADC_DATA_delay_min [expr 20.0*0.007]
set ADC_Tco_max      6.5
set ADC_Tco_min      2.5

set_input_delay -clock {adc_clk} -max [expr $ADC_CLK_delay_max + $ADC_Tco_max
+ $ADC_DATA_delay_max] [get_ports {adc_dat[*]}]

set_input_delay -clock {adc_clk} -min [expr $ADC_CLK_delay_min + $ADC_Tco_min
+ $ADC_DATA_delay_min] [get_ports {adc_dat[*]}]
```

О ужас, воскликнет кто-то, увидев кучу *[expr]* и много TCL переменных. Но мы то уже воробы стрельанные, давайте разбираться, что к чему.

```
set ADC_CLK_delay_max [expr 30.0*0.007]
set ADC_CLK_delay_min [expr 30.0*0.007]
```

Это задержка проводника тактовой частоты от ПЛИС до АЦП. Положим что на плате это проводник длиной 30мм, 0.007 нс/мм это оценочная задержка проводника 0.2мм на текстолите марки FR4. Можно было бы пересчитать в ручную, но лучше поручить эту работу TimeQuest.

```
set ADC_Tco_max 6.5
set ADC_Tco_min 2.5
```

А вот это уже параметры самого АЦП указанные в даташите. Я не использую типовое значение *Tco*, потому что нам нужно получить оценку для наихудшего случая. Теперь нам нужно прописать констрейны *tsu/th* для триггеров в ПЛИС:

```
set input_delay -clock {adc_clk} -max [expr $ADC_CLK_delay_max + $ADC_Tco_max
+ $ADC_DATA_delay_max] [get_ports {adc_dat[*]}]

set input_delay -clock {adc_clk} -min [expr $ADC_CLK_delay_min + $ADC_Tco_min
+ $ADC_DATA_delay_min] [get_ports {adc_dat[*]}]
```

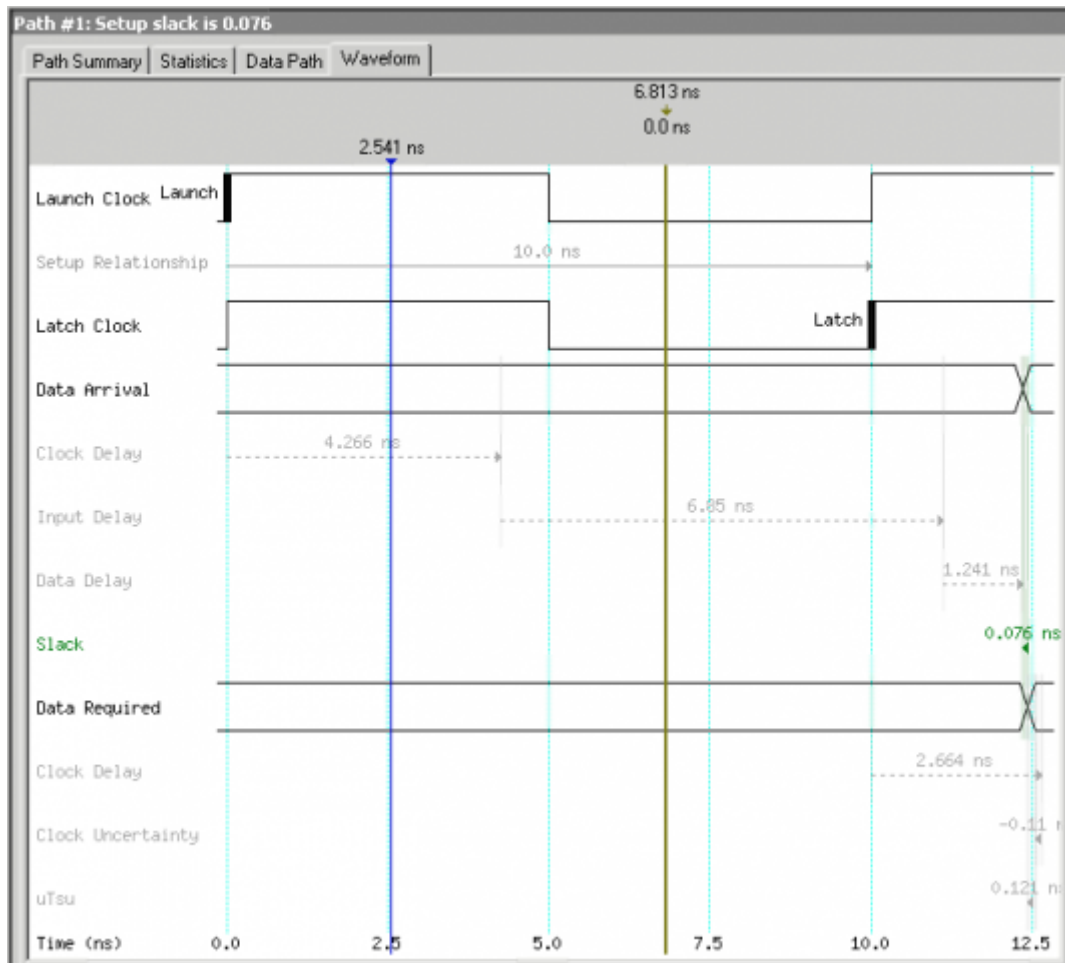
Видно, что в эти констрейны входят задержки и параметры АЦП, соответствующие наихудшему случаю. Собираем, запускаем, смотрим.

The screenshot shows the TimeQuest Timing Analyzer interface. The 'Report' window is open, displaying a tree view on the left with 'Inputs to Registers (Setup)' and 'Inputs to Registers (Hold)' selected. The main area shows two tables of timing data. The 'Inputs to Registers (Setup)' table lists 10 entries with slack values ranging from 0.076 to 0.095. The 'Inputs to Registers (Hold)' table lists 10 entries with slack values ranging from 5.223 to 5.242. The 'Tasks' window at the bottom shows a list of tasks including 'Report DDR', 'Report Metastability', 'Diagnostic', 'Report Clocks', 'Report Clock Transfers', 'Report Unconstrained Paths', 'Report SDC', 'Report Ignored Constraints', and 'Check Timing'.

Inputs to Registers (Setup)					
	Slack	From Node	To Node	Launch Clock	Latch Clock
1	0.076	adc_dat[2]	adc_io_reg[2]	adc_clk	clk
2	0.078	adc_dat[1]	adc_io_reg[1]	adc_clk	clk
3	0.081	adc_dat[0]	adc_io_reg[0]	adc_clk	clk
4	0.083	adc_dat[9]	adc_io_reg[9]	adc_clk	clk
5	0.088	adc_dat[7]	adc_io_reg[7]	adc_clk	clk
6	0.090	adc_dat[3]	adc_io_reg[3]	adc_clk	clk
7	0.091	adc_dat[8]	adc_io_reg[8]	adc_clk	clk
8	0.092	adc_dat[5]	adc_io_reg[5]	adc_clk	clk
9	0.092	adc_dat[6]	adc_io_reg[6]	adc_clk	clk
10	0.095	adc_dat[4]	adc_io_reg[4]	adc_clk	clk

Inputs to Registers (Hold)					
	Slack	From Node	To Node	Launch Clock	Latch Clock
1	5.223	adc_dat[4]	adc_io_reg[4]	adc_clk	clk
2	5.226	adc_dat[5]	adc_io_reg[5]	adc_clk	clk
3	5.226	adc_dat[6]	adc_io_reg[6]	adc_clk	clk
4	5.227	adc_dat[8]	adc_io_reg[8]	adc_clk	clk
5	5.228	adc_dat[3]	adc_io_reg[3]	adc_clk	clk
6	5.230	adc_dat[7]	adc_io_reg[7]	adc_clk	clk
7	5.234	adc_dat[9]	adc_io_reg[9]	adc_clk	clk
8	5.237	adc_dat[0]	adc_io_reg[0]	adc_clk	clk
9	5.240	adc_dat[1]	adc_io_reg[1]	adc_clk	clk
10	5.242	adc_dat[2]	adc_io_reg[2]	adc_clk	clk

Видим, что констрейны выполнены, но запас по ним распределен не равномерно. Особенно настораживает маленький запас по *tsu*. Делаем **Report Worst-Case Path** и давайте смотреть, в чем дело.



Мы видим на рисунке задержку от порта ПЛИС *clk* до порта ПЛИС *adc_clk* **Clock Delay = 4.266нс**, задержку сигнала на плате **Input Delay = 6.85нс** (недоверчивые могут посчитать так ли это, набрав на калькуляторе $(30+20)*0.007 + 6.5$) и задержку сигнала во входном I/O буфере **Data Delay = 1.241нс**. Все это составляет оценку прибытия данных в ПЛИС **Data Arrival**.

Теперь посмотрим на **Data Required**. Видим, что анализ задержки начинается от второго фронта **Latch Clock** на приемном триггере ПЛИС (так и должно быть для анализа *tsu*) и задержка от порта ПЛИС *clk* до тактового входа триггеров *adc_io_reg* **Clock Delay = 2.664нс**. И относительно этого момента времени оценивается выполнение *tsu* на входном триггере ПЛИС.

Как поступить в этом случае? Самый простой способ – использовать PLL и подвинуть клок, на котором работают триггеры в ПЛИС. Мы это уже делали, можете проделать это у себя на тестовом проекте. Рассмотрим другой вариант, как утоптать проект в констрейны, который часто используется – метод мультицикловых путей.

Для этого инвертируем клок:

```
assign adc_clk = ~clk;
```


И соответственно поправим sdc файл:

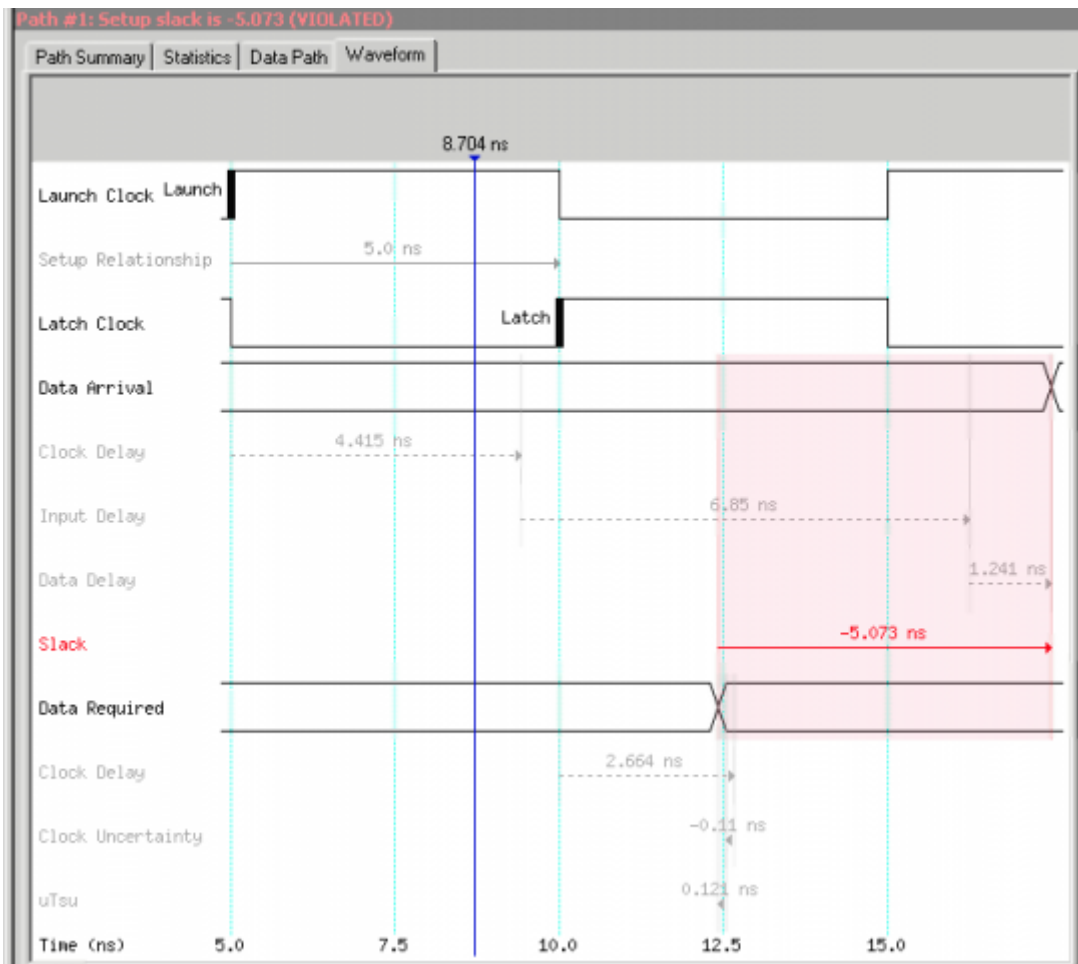
```
create_generated_clock -name {adc_clk} -source [get_ports {clk}] -invert [get_ports {adc_clk}]
```

Все остальное остается тем же. Собираем, проверяем и в итоге:

Inputs to Registers (Setup)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	-5.073	adc_dat[2]	adc_io_reg[2]	adc_clk	clk	
2	-5.071	adc_dat[1]	adc_io_reg[1]	adc_clk	clk	
3	-5.068	adc_dat[0]	adc_io_reg[0]	adc_clk	clk	
4	-5.066	adc_dat[9]	adc_io_reg[9]	adc_clk	clk	
5	-5.061	adc_dat[7]	adc_io_reg[7]	adc_clk	clk	
6	-5.059	adc_dat[3]	adc_io_reg[3]	adc_clk	clk	
7	-5.058	adc_dat[8]	adc_io_reg[8]	adc_clk	clk	
8	-5.057	adc_dat[5]	adc_io_reg[5]	adc_clk	clk	
9	-5.057	adc_dat[6]	adc_io_reg[6]	adc_clk	clk	
10	-5.054	adc_dat[4]	adc_io_reg[4]	adc_clk	clk	

Inputs to Registers (Hold)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	10.366	adc_dat[4]	adc_io_reg[4]	adc_clk	clk	
2	10.369	adc_dat[5]	adc_io_reg[5]	adc_clk	clk	
3	10.369	adc_dat[6]	adc_io_reg[6]	adc_clk	clk	
4	10.370	adc_dat[8]	adc_io_reg[8]	adc_clk	clk	
5	10.371	adc_dat[3]	adc_io_reg[3]	adc_clk	clk	
6	10.373	adc_dat[7]	adc_io_reg[7]	adc_clk	clk	
7	10.377	adc_dat[9]	adc_io_reg[9]	adc_clk	clk	
8	10.380	adc_dat[0]	adc_io_reg[0]	adc_clk	clk	
9	10.383	adc_dat[1]	adc_io_reg[1]	adc_clk	clk	
10	10.385	adc_dat[2]	adc_io_reg[2]	adc_clk	clk	

Видим, что все в слагах. Посмотрим внимательно, что же произошло.



Мы видим, что точка отсчета *tsu* сдвинулась на 5нс влево, что естественно, привело к ломке констейнов. Для того, чтобы эта картина стала больше похоже на действительность, нужно передвинуть точку отсчета на второй фронт частоты **Latch Clock**. Для этого указываем TimeQuest что при анализе времянок нужно учитывать этот сдвиг.

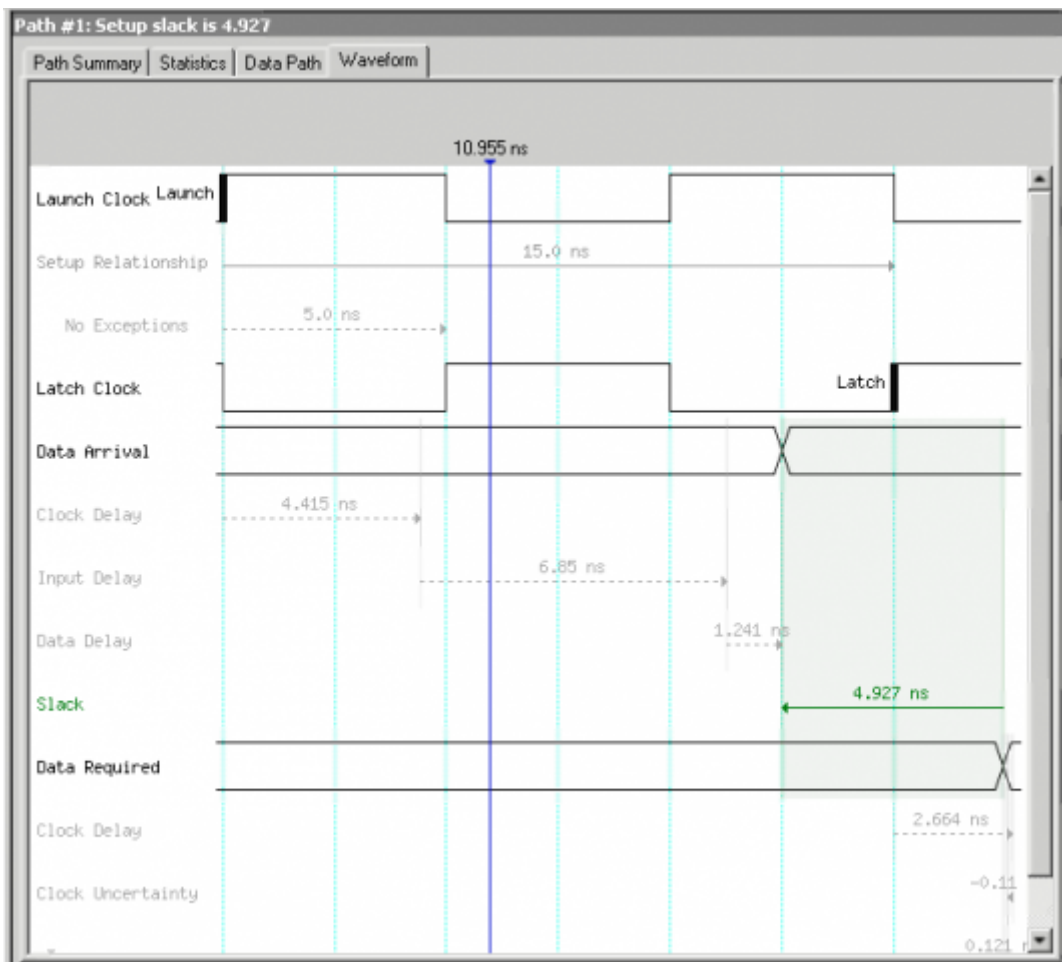
```
set_multicycle_path -end -from {get_clocks {adc_clk}} -to {get_clocks {clk}} -setup 2
```

Т.е. я указал клок источник **-from adc_clk**, клок приемник **-to clk**, указал, что анализ идет по **setup** и то, что нужно учитывать второй фронт клона. А то, что этот фронт нужно **применить именно к приемному клок**у я указал с помощью ключа **-end**.

Теперь сбрасываем проект, запускаем анализ заново и

Report						
<ul style="list-style-type: none"> TimeQuest Timing Analyzer Summary SDC File List Report Timing (I/O) <ul style="list-style-type: none"> Inputs to Registers (Setup) Inputs to Registers (Hold) 						
Inputs to Registers (Setup)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	4.927	adc_dat[2]	adc_io_reg[2]	adc_clk	clk	
2	4.929	adc_dat[1]	adc_io_reg[1]	adc_clk	clk	
3	4.932	adc_dat[0]	adc_io_reg[0]	adc_clk	clk	
4	4.934	adc_dat[9]	adc_io_reg[9]	adc_clk	clk	
5	4.939	adc_dat[7]	adc_io_reg[7]	adc_clk	clk	
6	4.941	adc_dat[3]	adc_io_reg[3]	adc_clk	clk	
7	4.942	adc_dat[8]	adc_io_reg[8]	adc_clk	clk	
8	4.943	adc_dat[5]	adc_io_reg[5]	adc_clk	clk	
9	4.943	adc_dat[6]	adc_io_reg[6]	adc_clk	clk	
10	4.946	adc_dat[4]	adc_io_reg[4]	adc_clk	clk	
Inputs to Registers (Hold)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	0.366	adc_dat[4]	adc_io_reg[4]	adc_clk	clk	
2	0.369	adc_dat[5]	adc_io_reg[5]	adc_clk	clk	
3	0.369	adc_dat[6]	adc_io_reg[6]	adc_clk	clk	
4	0.370	adc_dat[8]	adc_io_reg[8]	adc_clk	clk	
5	0.371	adc_dat[3]	adc_io_reg[3]	adc_clk	clk	
6	0.373	adc_dat[7]	adc_io_reg[7]	adc_clk	clk	
7	0.377	adc_dat[9]	adc_io_reg[9]	adc_clk	clk	
8	0.380	adc_dat[0]	adc_io_reg[0]	adc_clk	clk	
9	0.383	adc_dat[1]	adc_io_reg[1]	adc_clk	clk	
10	0.385	adc_dat[2]	adc_io_reg[2]	adc_clk	clk	
Tasks						
<ul style="list-style-type: none"> Report DDR Report Metastability Dagnostic <ul style="list-style-type: none"> Report Clocks Report Clock Transfers Report Unconstrained Paths Report SDC Report Ignored Constraints 						

видим, что слаки ушли и запасы по th стали больше, чем были запасы по tsu до этого. Хотя запас все же маловат. Но нам в этом случае важен сам метод. Посмотрим подробнее, как идет анализ:



Как видим, точка отсчета сдвинулась на второй фронт. Как и было запланировано.

Вот собственно и все про данный вид интерфейсов. Как видите ничего сложного, нужно только обязательно учитывать разводку интерфейсов на печатной плате.

Source-Synchronous Input

Вот мы и подходим к последнему типу рассматриваемых интерфейсов. Как мы уже рассматривали в таких интерфейсах, тактовая частота идет от отдельного генератора или источника. И в данном случае описание констрейнов идет через виртуальный блок. Рассмотрим опять наш АЦП:

```
module adc (input clk, input [9 : 0] adc_dat, output logic [9 : 0]
data);
```

```
logic [9 : 0] adc_io_reg;
logic [9 : 0] adc_reg;
```

```
always_ff@(posedge clk) begin
    {adc_reg, adc_io_reg} <= {adc_io_reg, adc_dat};
end
```

```
always_ff@(posedge clk) begin
    data <= adc_reg;
end
```

endmodule

sdc файл в этом случае будет следующий:

set_time_format -unit ns -decimal_places 3

derive_clock_uncertainty

create_clock -period 100MHz -name {clk} [get_ports {clk}]

create_clock -period 100MHz -name {virt_clk}

set_clock_groups -exclusive -group {clk virt_clk}

*set CLK_as_delay_max [expr 30.0*0.007]*

*set CLK_as_delay_min [expr 30.0*0.007]*

*set CLK_ad_delay_max [expr 30.0*0.007]*

*set CLK_ad_delay_min [expr 30.0*0.007]*

*set ADC_DATA_delay_max [expr 20.0*0.007]*

*set ADC_DATA_delay_min [expr 20.0*0.007]*

set ADC_Tco_max 6.5

set ADC_Tco_min 2.5

set_input_delay -clock {virt_clk} -max [expr \$CLK_as_delay_max + \$ADC_Tco_max + \$ADC_DATA_delay_max - \$CLK_ad_delay_max] [get_ports {adc_dat[]}]*

set_input_delay -clock {virt_clk} -min [expr \$CLK_as_delay_min + \$ADC_Tco_min + \$ADC_DATA_delay_min - \$CLK_ad_delay_min] [get_ports {adc_dat[]}]*

Как вы видите, sdc файл аналогичен тому, что использовался для Source Synchronous Output, и построен по тем же правилам. Собираем, запускаем анализ и,

Inputs to Registers (Setup)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	2.638	adc_dat[4]	adc_io_reg[4]	virt_clk	clk	
2	2.639	adc_dat[1]	adc_io_reg[1]	virt_clk	clk	
3	2.640	adc_dat[8]	adc_io_reg[8]	virt_clk	clk	
4	2.645	adc_dat[3]	adc_io_reg[3]	virt_clk	clk	
5	2.647	adc_dat[7]	adc_io_reg[7]	virt_clk	clk	
6	2.648	adc_dat[5]	adc_io_reg[5]	virt_clk	clk	
7	2.654	adc_dat[0]	adc_io_reg[0]	virt_clk	clk	
8	2.654	adc_dat[9]	adc_io_reg[9]	virt_clk	clk	
9	2.656	adc_dat[2]	adc_io_reg[2]	virt_clk	clk	
10	2.656	adc_dat[6]	adc_io_reg[6]	virt_clk	clk	

Inputs to Registers (Hold)						
	Slack	From Node	To Node	Launch Clock	Latch Clock	
1	2.664	adc_dat[2]	adc_io_reg[2]	virt_clk	clk	
2	2.665	adc_dat[6]	adc_io_reg[6]	virt_clk	clk	
3	2.667	adc_dat[0]	adc_io_reg[0]	virt_clk	clk	
4	2.667	adc_dat[9]	adc_io_reg[9]	virt_clk	clk	
5	2.672	adc_dat[5]	adc_io_reg[5]	virt_clk	clk	
6	2.674	adc_dat[7]	adc_io_reg[7]	virt_clk	clk	
7	2.675	adc_dat[3]	adc_io_reg[3]	virt_clk	clk	
8	2.680	adc_dat[8]	adc_io_reg[8]	virt_clk	clk	
9	2.681	adc_dat[1]	adc_io_reg[1]	virt_clk	clk	
10	2.683	adc_dat[4]	adc_io_reg[4]	virt_clk	clk	

как видите, все в шоколаде %).

Вот мы с вами и рассмотрели самую большую и требующую тонкой настройки и шаманства область приложения TimeQuest. Некоторые могут сказать "зачем нам что-то констрейнить, и так сойдет", что ж, это их право. Я же думаю, что знать, что твое устройство будет работать во всех **Worst Case** без ошибок по времянке лучше, чем сидеть и ждать пока где-то эта ошибка выстрелит %)

TimeQuest для чайников. Часть 5 (Заключение)

Мы рассмотрели основные вопросы по заданию констрейнов для различных проектов.

За границами рассмотрения остались такие виды задания ограничений, как мультицикловые пути в проекте (мы рассмотрели их использование только в интерфейсах), интерфейсы с работой по обоим фронтам.

Цель охватить все возможные ситуации не ставилась, предоставленной информации достаточно для начала использования TimeQuest. То, что будет не понятно, можно найти в документации на TimeQuest и в хендбуке на квартус. Все это доступно на сайте альтеры.

Если же возникнут вопросы, ответы на которые вы не нашли, то задать вы их можете на форум www.electronix.ru. Или вы можете отправить вопрос мне в почту. Но заранее предупреждаю, вопрос должен быть предметный, иначе письмо полетит в мусор. В письме должно быть следующее:

1. Тема письма - Вопрос/проблема в TimeQuest
2. Описание проблемы - что у вас не получается

3. Вычлененная проблема из вашего проекта - маленький модуль, который демонстрирует логику вашего проекта, где есть затык. К этому файлу qsf/qpj файлы проекта.
4. Ваше решение проблемы - ваш sdc файл, в котором должно быть такое задание констрейнов, которое вы считаете нужным. Заранее предупреждаю, если в sdc файле будут только клоки и не будет, например описания констрейнов ввода-вывода, а вопрос вы задаете именно про них, на такие письма отвечать не буду. Прочитайте еще раз блог, как задавать эти констрейны, я описал.

Ответ на вашу проблему будет оформлен как приложение к блогу "TimeQuest для чайников".

Напоследок еще немного полезных док про TimeQuest

1. Constraining_SOPC_Designs_v11.0.doc - обзорная дока о задании констрейнов для дизайнов в SOPC Builder
2. mnl_sdctmq.pdf - справочник по командам TimeQuest a

Желаю удачи в освоении TimeQuest a, вы уже убедились, что в этом нет ничего сложного %).