Synopsys Design Constraint — язык задания временных ограничений на примере Altera TimeQuest. Часть 2

Денис ШЕХАЛЕВ
shdv@micran.ru

В предыдущей части мы рассмотрели основы временного анализа с помощью TimeQuest на примере простого проекта, состоящего из одного счетчика. В том проекте был всего один домен тактовой частоты. Но большинство проектов содержат большее количество доменов тактовых частот, с различными механизмами взаимодействия между ними. В этой части мы рассмотрим типовые проекты с несколькими тактовыми доменами и покажем, как правильно задать временные ограничения для таких проектов. Как вы увидите, это ненамного сложнее, чем для одночастотного проекта, рассмотренного нами ранее.

Перед тем как начать рассмотрение проектов, хотелось бы отметить следующее: в этой части статьи будут рассматриваться только вопросы задания временных ограничений тактовых частот. Задание временных ограничений для интерфейсов ввода/вывода мы обсудим в следующих частях.

Несколько тактовых частот, идущих снаружи ПЛИС

Рассмотрим простой ШИМ-модулятор с управлением по 3-проводному интерфейсу SPI:

```
module spi (input clk, cs_n, sdi, sclk, output logic led ) ;
   //-----------------------------------------------------------
   // spi clock domain
   //-----------------------------------------------------------
   logic [15 : 0] sdata;

   always_ff @(posedge sclk) begin // spi shift register
      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 detector

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

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

endmodule
```

С учетом полученных нами ранее знаний составим sdc-файл для этого проекта. Как мы видим, в этом проекте используется две тактовые частоты — ***clk*** и ***sclk***. Значит, нам нужно их описать. Положим, что частота ***clk*** получена с тактового генератора, а ***sclk*** — с управляющего процессора, и частоты равны 100 и 10 МГц соответственно:

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

По умолчанию TimeQuest считает, что все тактовые частоты, описанные в sdc-файле, синхронизированы друг с другом. Это означает, что анализ временных ограничений в пути передачи данных начинается с точки, местоположение которой определяется начальными смещениями, определенными в команде ***create_clock***. В нашем конкретном случае при временном анализе передние фронты частот ***clk*** и ***sclk*** будут совпадать. Это не соответствует действительности, так как частоты получены от разных генераторов.

Внимательно посмотрим на код. Видим синхронизатор сигнала управления захвата SPI-данных, выполненный на регистре ***cs_reg***. Благодаря этому синхронизатору к моменту захвата SPI-данных, сформированных в домене ***sclk***, в домене ***clk*** данные будут неизменны в течение двух тактовых интервалов этого домена. Это предположение верно только в том случае, если управляющий процессор не начнет тут же новую транзакцию. Но даже в этом случае при отношении ***clk/sclk*** в 10 раз ПЛИС успеет захватить данные.

С учетом сделанных выводов можно заключить, что анализировать путь передачи данных между доменами не нужно. Описать это временное ограничение в sdc-файле можно 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 любые регистры, тактируемые частотами ***sclk*** и ***clk***:

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

3. Описав группы логических тактовых сигналов, которые являются асинхронными (эксклюзивными) друг другу:

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

4. Описав каждый логический тактовый сигнал как асинхронный (эксклюзивный) к любому другому тактовому сигналу:

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

Все, задание временных ограничений для тактовых частот проекта завершено. Мы описали все тактовые частоты и указали их взаимосвязь друг с другом. Теперь Quartus может оптимизировать логику в тактовых доме-

нах независимо друг от друга. И при анализе выполнения временных ограничений пути между доменами рассматриваться не будут.

Но нужно отметить, что все эти способы не эквивалентны друг другу. При использовании первого способа не будет анализироваться конкретный путь между регистрами ***sdata/level***, второго — все пути из домена ***sclk*** в домен ***clk***, а в третьем и четвертом — все пути из домена ***sclk*** в домен ***clk*** и **наоборот**. Какой именно метод задания игнорируемых путей использовать, определяется контекстом проекта и приоритетом разработчика. Автор считает, что четвертый способ задания асинхронных групп более предпочтительный, так как позволяет более просто задать отношение тактового сигнала ко всем другим тактовым сигналам проекта.

Внимательные разработчики отметят недоработку в этом способе задания временных ограничений для пути данных между доменами. Действительно, если путь данных между регистрами ***sdata/level*** будет больше двух периодов частоты ***clk*** (в нашем примере 20 нс), то схема работать не будет. Поэтому нужно добавить к sdc-файлу задание максимальной задержки пути между регистрами с помощью команды ***set_max_delay***:

```
set_max_delay -from [get_registers {sdata[*]}] -to [get_registers {level[*]}] 10.0
```

Но для ПЛИС фирмы Altera этого можно не делать, потому что Quartus при синтезе и разводке старается минимизировать задержки, и это ограничение для большинства случаев будет выполнено автоматически.

Тактовые частоты, порожденные логикой, находящейся 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 клока.

Порождение новой тактовой частоты без PLL

Рассмотрим простой модуль:

```
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
```

Как видите, для генерации тактовой частоты 50 МГц используется делитель на счетном триггере ***ff***. Сигнал с этого делителя подается на глобальную тактовую линию, на которой реализован счетчик для вывода индикации на светодиод. Тактовые частоты, полученные с помощью таких преобразований, описываются так:

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

Мы задали логическое имя тактовой частоты после преобразования (***clk_50MHz***), источник тактовой частоты до преобразования (***clk_100MHz***), формулу преобразования частот (***divide_by 2***) и физический объект (регистр ***ff***), который является источником частоты ***clk_50MHz***. Обратите особое внимание на то, что источником тактовой частоты для преобразования указывается не логическое имя тактовой частоты, а именно его физический источник. В нашем примере это порт ПЛИС ***clk_100MHz***.

С учетом всех ранее полученных нами знаний sdc-файл для этого проекта 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 новой тактовой частоты с помощью PLL

Рассмотрим простой перемежитель входных данных для ЦАП, так называемый маппер:

```
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 перемежения нужно мультиплексировать данные на выходе ПЛИС на частоте, которая в два раза больше частоты поступающих входных символов. Первое, что нужно сделать, это описать умножение частоты. Второе — описать соотношения тактовых частот, поместив их в одну группу, потому что в этом 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 задать все параметры преобразования частоты, ее логическое имя и источник новой частоты (порт PLL):

```
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 этого подхода является то, что при изменении коэффициентов преобразования частоты PLL нужно изменять строку в sdc-файле.
2. С помощью команды ***derive_pll_clocks***. Что происходит при выполнении этой команды, можно посмотреть в консоли TimeQuest. 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 видите, описание новой тактовой частоты создалось автоматически, с учетом всех преобразований. Минусом этого подхода является то, что тактовая частота получает длинное логическое имя. Описание отношений клоков, соответственно, будет менее элегантное:

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

Но, как мы помним, sdc-файл — это TCL-скрипт. Следовательно, можно воспользоваться TCL-переменными. В этом случае описание отношений тактовых частот будет таким:

```
set clk_x2 pll|altpll_component|auto_generated|pll1|clk[0]
set_clock_groups -exclusive -group [list $clk_x2 iclk]
```

Тактовые частоты, порожденные логикой, находящейся вне ПЛИС

Рассмотрим пример простой системы, в которой одна из тактовых частот порождается вне ПЛИС (рис. 14). Вне ПЛИС находится делитель, который является источником частоты в 100 МГц. Данные передаются из тактово-

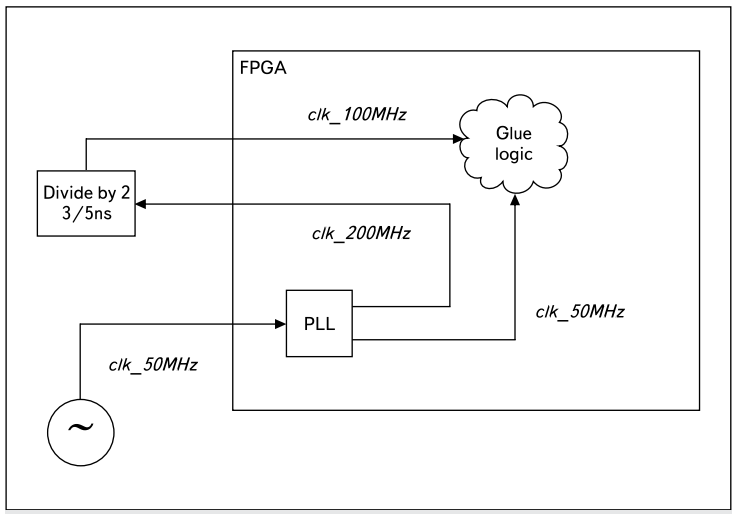

Рис. 14. Структурная схема системы, в которой одна из тактовых частот порождается вне ПЛИС

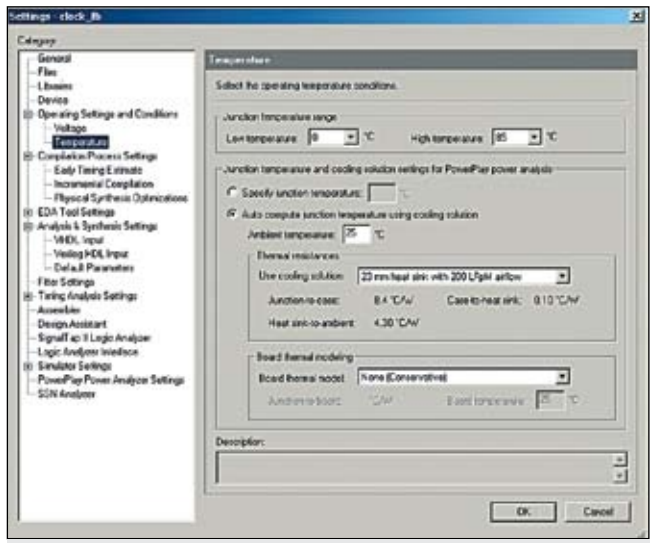

Рис. 16. Диалоговое окно Quartus для задания режимов работы ПЛИС

го домена 100 МГц в тактовый домен 50 МГц, поэтому для корректного задания временных ограничений нужно учесть задержки вне ПЛИС, которые TimeQuest рассчитать не может, а именно задержку сигнала на делителе частоты и задержку сигнала на проводниках печатной платы. Для примера предположим, что разброс задержки делителя частоты равен 3–5 нс, а задержка тактового сигнала на проводниках ПП составляет 0,2 нс.

Описание подобной системы на языке SystemVerilog следующее:

```
module clock_fb (input iclk50MHz, iclk100MHz, idat, output
oclk200MHz, odat);

  logic clk50MHz;

  pll pll (iclk50MHz, clk50MHz, oclk200MHz);

  logic [1 : 0] dat;

  always_ff @(posedge iclk100MHz) begin
    dat <= (dat << 1) | idat;
  end

  always_ff @(posedge clk50MHz) begin
    odat <= dat[1];
  end

endmodule
```

Для этого проекта sdc-файл будет такой:

```
# base clocks
derive_clock_uncertainty
create_clock -name clk50MHz -period 50MHz [get_ports
{iclk50MHz}]
# generated clocks
derive_pll_clocks
set clk50MHz pll|altpll_component|auto_generated|pll1|clk[0]
set clk200MHz pll|altpll_component|auto_generated|pll1|clk[1]
# output path of feeback
create_generated_clock -name oclk200MHz -source $clk200MHz
[get_ports {oclk200MHz}]
# input path of feeback
create_generated_clock -name clk100MHz -source [get_ports
{oclk200MHz}] -divide_by 2 [get_ports {iclk100MHz}]
set_clock_latency -source 4.2 [get_clocks {clk100MHz}]
# used uncertanty
set_clock_uncertainty -to [get_clocks clk100MHz] 1
set_clock_uncertainty -from [get_clocks clk100MHz] -to [get_clocks
$clk50MHz] 1
# synchronus transfers
set_clock_groups -exclusive -group [list $clk50MHz clk100MHz]
```

Рассмотрим те места sdc-файла, на которые нужно обратить внимание:

```
create_generated_clock -name oclk200MHz -source $clk200MHz
[get_ports {oclk200MHz}]
```

Здесь все понятно. Описываем тактовую частоту в 200 МГц, которую выдаем наружу через порт ПЛИС:

```
create_generated_clock -name clk100MHz -source [get_ports
{oclk200MHz}] -divide_by 2 [get_ports {iclk100MHz}]
set_clock_latency -source 4.2 [get_clocks {clk100MHz}]
```

В этих строках мы описываем внешний делитель частоты. И задаем задержку на этом делителе в 4,0 + 0,2 нс. Если возникновение значения 0,2 нс не вызывает вопросов, то по поводу другого слагаемого возникает логичный вопрос: почему именно 4 нс? Ответ заключен в этих строках:

```
set_clock_uncertainty -to [get_clocks clk100MHz] 1.0
set_clock_uncertainty -from [get_clocks clk100MHz] -to [get_clocks
$clk50MHz] 1.0
```

То есть мы задали среднюю задержку на делителе 4,0 нс и нестабильность тактовой частоты на этом делителе в 1 нс. Таким способом мы описали разброс задержки делителя частоты в 3–5 нс. Все, учет всех остальных задержек Quartus и TimeQuest сделают сами.

Собираем проект и видим результат временного анализа (рис. 15). Есть ошибки, запускаем TimeQuest и смотрим, где именно происходит нарушение. Но на выполнение команды ***Report Top Falling Paths*** TimeQuest пишет в консоли следующее:

```
Info: No failing paths found
```

Дело в том, что на рис. 15 видно нарушение только в одном из режимов анализа —

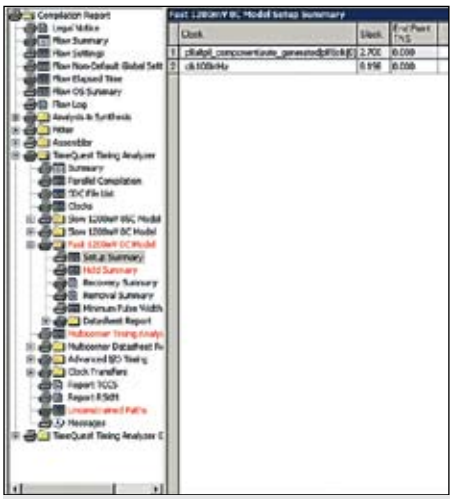

Рис. 15. Результат временного анализа многочастотного проекта с нарушением временных ограничений в одном из режимов

Fast 1200m V 0C Model. Эта модель соответствует самым быстрым экземплярам используемой нами ПЛИС. А ***0C*** означает, что анализ проводился для минимальной температуры использования изделия, определенной в Settings -> Operating Settings and Conditions (рис. 16).

Устанавливаем нужный нам режим анализа в TimeQuest (рис. 17) и смотрим, где нарушаются заданные временные ограничения. На рис. 18 видно, что нарушается условие по ***th*** из-за того, что ***DataDelay*** слишком мала. Для устранения этого нарушения достаточно указать Quartus оптимизировать разводку под все возможные случаи (corner-case) (рис. 19). Собираем проект и видим, что Quartus увеличил задержки за счет разводки (рис. 20), убрав, таким образом, нарушение временного ограничения.

Внимание. *Не относитесь халатно к полноте временного анализа и точному заданию условий работы ПЛИС. Приведенный пример*

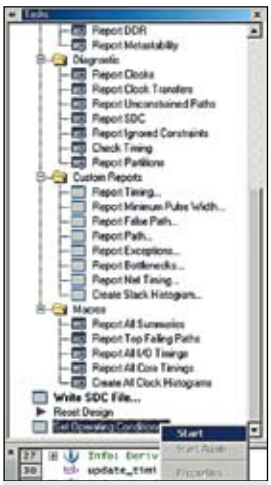

Рис. 17. Смена режима временного анализа в TimeQuest

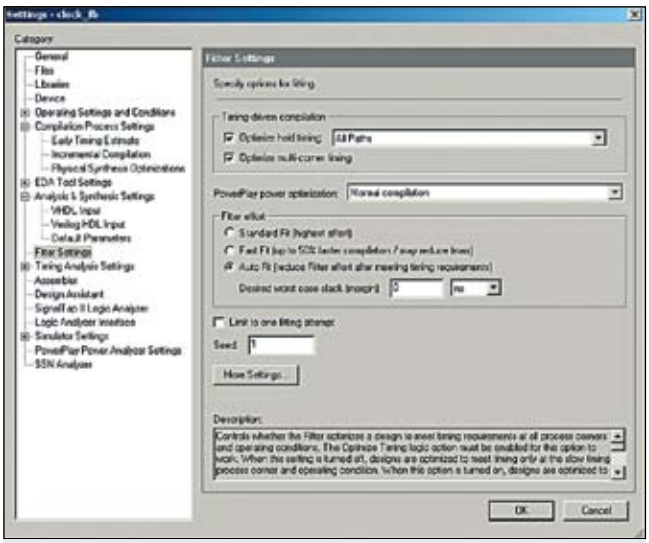

Рис. 19. Диалоговое окно Quartus для задания опций разводки

демонстрирует, что это важно. Задав некорректные условия работы ПЛИС, можно долго воевать с выполнением временных ограничений или ждать, пока Quartus будет перебирать разные варианты разводки проекта, чтобы их выполнить.

Тактовые частоты, мультиплексируемые внутри

Иногда требуется сделать мультиплексор тактовой частоты внутри ПЛИС. В этом случае при разводке и временном анализе Quartus и TimeQuest должны учесть задержку сигнала тактовой частоты на мультиплексоре и обеспечить выполнение временных соотношений триггеров, которые работают на мультиплексированном сигнале. Рассмотрим пример простого мультиплексора. Положим, что глитч, неизбежно возникающий при таком описании мультиплексора тактовых частот, не важен:

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

Для такого проекта sdc-файл 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}
set_clock_groups -exclusive -group {clk2}
```

Командой описания отношения тактовых частот мы задаем то, что частоты ***clk1*** и ***clk2*** представляют собой эксклюзивные группы, то есть цепь ***mux_clk*** рассматривается нагруженной либо тактовой частотой ***clk1***, либо ***clk2***. Все, больше ничего не нужно описывать, остальное Quartus и TimeQuest определят и сделают самостоятельно.

Тактовые частоты, мультиплексируемые снаружи

Есть системы, когда на один и тот же тактовый вход ПЛИС могут подаваться разные тактовые частоты. Для того чтобы корректно задать временные ограничения в таком случае, нужно создать несколько логических тактовых частот, идущих с одного физического источника (порта ПЛИС). Сделать это можно следующим образом:

```
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
```

Все, анализ выполнения временных ограничений будет произведен по всем тактовым частотам автоматически.

Заключение

Мы рассмотрели основные типы многочастотных проектов для ПЛИС. Вне зависимости от количества частот для любого проекта последовательность действий при разработке sdc-файла одна и та же:

1. Описать все тактовые частоты, приходящие снаружи ПЛИС (***create_clock***).
2. Описать все тактовые частоты, исходящие из ПЛИС (***create_generated_clock***).
3. Описать все тактовые частоты, генерируемые внутри/снаружи ПЛИС (***create_generated_clock/derive_pll_clocks***).

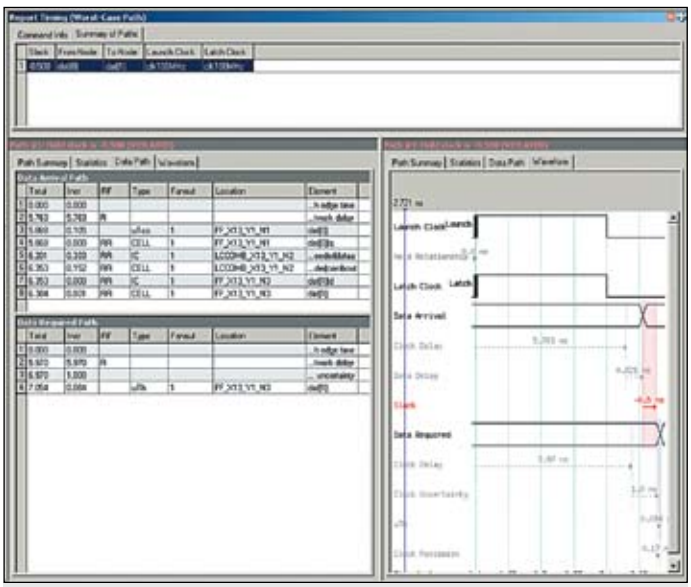

Рис. 18. Временной анализ проекта с внешним делителем частоты с нарушением по th

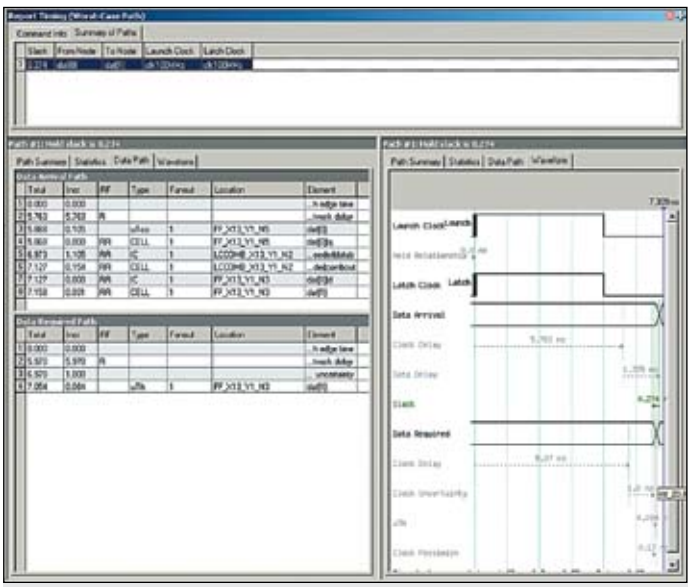

Рис. 20. Временной анализ проекта с внешним делителем частоты с устраненным нарушением по th

4. Описать отношения тактовых частот, определив их в асинхронные/эксклюзивные группы (***set_clock_groups***).

Если вы будете придерживаться этого алгоритма, у вас никогда не будет проблем с заданием временных ограничений для тактовых частот проекта. В следующей статье будут рассмотрены интерфейсы ввода/вывода. Правильное задание временных ограничений интерфейсов вызывает наибольшее количество вопросов у начинающих пользователей TimeQuest и представляет собой своего рода некоторый элемент «черной магии» TimeQuest. Но как вы увидите, в этом нет ничего сложного.

Литература

1. Quartus II Handbook Version 9.0 — http://www.altera.com/literature/lit-qts.jsp
2. SDC and TimeQuest API Reference Manual — http://www.altera.com/literature/manual/mnl_sdctmq.pdf
3. Quartus II TimeQuest Timing Analyzer Cookbook — http://www.altera.com/literature/manual/mnl_timequest_cookbook.pdf
4. Шехалев Д. Synopsys Design Constraint — язык задания временных ограничений на примере Altera TimeQuest. Часть 1 // Компоненты и технологии. 2010. № 9.
5. http://embedders.org/blog/des00