Specifying Timing Constraints

In addition to ensuring that your design produces the correct outputs, you also need to ensure that various timing requirements will be met. This lecture describes how to describe these requirements in the form of statements in SDC (Synopsys Design Constraint) files.
After this lecture you should be able to use the SDC constraints described in this lecture to check that your design: (a) operates at specified clock rate(s), (b) meets the setup and hold time requirements of external IO devices, and (c) operates with the specified delays of external devices.

Introduction

Ensuring that the timing requirements ("constraints") of a design are met is often as challenging as ensuring the design is logically correct.

Timing constraints can come from system requirements (e.g. a required clock rate resulting from a required throughput or response time requirements), or from the timing requirements (e.g. the setup/hold times) and specifications (e.g. propagation delays) of other components.

A [static] timing analyzer is a program that analyzes a netlist that includes propagation delays and checks that specified timing constraints are met. The designer's job is to specify these timing constraints correctly and modify the design as necessary until the constraints are met (usually while also trying to minimize costs and meeting deadlines).

There is a standard file format, Synopsys Design Constraint (SDC), for specifying timing and other design constraints although some tools may not include all commands and sometimes add their own commands. This lecture covers some of the more common timing constraint and how they are specified in an SDC file.

Note that adding timing constraints does not change the architecture of a design. The designer must understand the speed limitations of the devices being used and the architecture that results from the HDL to come up with a design that is likely to meet the constraints.

Collections

We often want to set timing constraints on a collection of signals such as all of the bits in a bus.

When a module is instantiated unique signal names must be generated to avoid name conflicts. These names may be difficult to determine. For example the signal name `adcspi:a0|cnt.bitcnt[0]` may specify the 0'th bit of the `bitcnt` field of the `cnt` signal of the `a0` instance of the `adcspi` module.

SDC files can use tcl functions that look up signal names by matching netlist names with patterns that can include the an asterisk as a wildcard. For example, the `[get_nodes *bitcnt[*]` would return references to the matching signal names, including the one above.

Different collection `get_` functions can search for different type of signals[1]: `nodes` most netlist items, `cells` (registers), `pins` (cell i/o's), `nets` (connections between pins), `ports` (top-level i/o), `clocks` (clock signals – not necessarily in your design). For example, `get_ports a*` would return references to all top-level IO signals matching the pattern "a*."

The netlist viewer can be used to look up node names from a generated schematic. Sometimes it's useful to get a list of the signal names in a collection. The following example shows tcl code that does this:

```
set c [get_ports *]
foreach_in_collection i $c { puts [get_port_info -name $i]}
```

Clock Constraints

Most designs will use one (or more) clock signals. These clocks' frequencies must be specified in the SDC file so that the timing analyzer can ensure that the setup and hold times of flip-flops internal to the FPGA will be met. Many designs will also derive new clock signals either by dividing a clock or by using a phase-locked loop (PLL).

You can use the following tcl commands in the SDC file to supply clock constraints. The syntax de-

[1] From Quartus documentation

tails are available in the Quartus SDC editor (right-click and select Insert Constraint), the TimeQuest timing analyzer GUI (Constraints...), and the Quartus documentation.

create_clock - this specifies a clock signal. The frequency is specified, it is given a name and it can be associated with a particular net in the design. For example:

```
create_clock -name clk50 -period 20 \
      [get_ports {clk50}]
```

create_generated_clock - can be used to define clocks with a fixed relationship (frequency and/or phase) to another clock. For example:

```
create_generated_clock \
   -source [get_ports clk50]\
   -divide_by 2 -name sclk_int \
   [get_nodes sclk~reg0]
```

derive_pll_clocks - this command uses PLL instance parameters to create generated clocks corresponding to the PLL output(s). Most FPGA designs use PLLs to generate clocks and this is often the only clock constraint required.

derive_clock_uncertainty - this adds the effect of clock (typically PLL) jitter to the timing analysis. Quartus gives an annoying warning if you do not use this constraint.

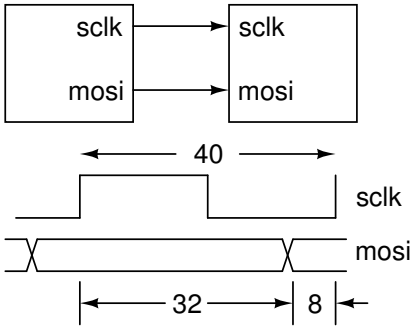
These constraints are all that are required for simple designs that do not use clocked I/O (e.g. LEDs and switches).

IO Timing Constraints

Most other timing constraints (e.g setup and hold times) are specified relative to clock signals. Many FPGA designs will therefore include the following SDC statements:

set_output_delay - this command allows you to specify the maximum or minimum allowed delay between an output clock and a data output. Use this to ensure the timing requirements of other devices will be met. You can use this to specify the setup (with the `-max` option) and/or hold (with `-min`) requirements of an off-chip control register or memory.

For example, if an external IC has an 8 ns minimum setup time between its `mosi` data input and `sclk` clock inputs and the clock period is 40 ns then the maximum delay from the clock to the data being valid is $40 - 8 = 32$ ns:



We could specify this as follows[2].

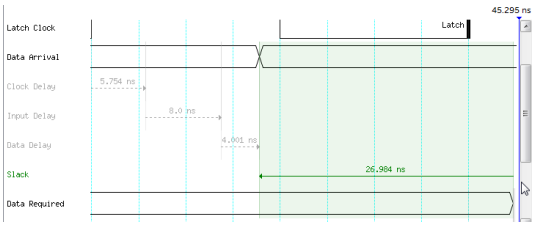
```
set_output_delay -clock sclk \
      -max 32 [get_ports mosi]
```

set_input_delay - this command allows you to specify the maximum or minimum delay between a clock output and a data input. Use this to ensure your circuit will work correctly given the output timing specifications of other devices. For example, you can use this to specify the clock-to-output (access) and propagation delays of an off-chip status register or memory.

For example if an external IC has an 8 ns delay from its `sclk` input to its `miso` output being valid, we could specify the constraint:

```
set_input_delay -clock sclk \
      -max 8 [get_ports miso]
```

The following diagram (from a Timing Analyzer report) shows that the analysis includes the clock propagation delays to the launch and latching flip-flops and the external 8 ns delay:



[2] This ignores propagation delays due to PCB traces. Velocity of propagation depends on the substrate but will be approximately $c/2$ or about 30 cm/ns

Different types of I/O interfaces require different types of timing constraints:

Common-Clock At low clock rates (10's of MHz) a common system clock usually supplies timing for interfaces. An example would be static RAM and older versions of the PCI bus. In this case the source clock is typically an input (external to the FPGA).

Source-Synchronous Source-synchronous interfaces supply a clock along with the data. Examples include SDRAM and SPI interfaces. In this case (as above) the source clock is generated by the FPGA when it's an interface master and is an input if the FPGA is an interface slave.

Embedded-Clock At higher rates (GHz) it's common to recover the clock from the data. Examples include USB and recent versions of HDMI. These types of interfaces require custom interface hardware (called SERDES – Serializer/Deserializer) to recover the clock from the data.

Exceptions

There are various situations that require additional constraints. See the timing analyzer documentation for more details.

Asynchronous Clocks

By default all clocks are assumed to be derived from the same master clock and retain the same timing relative to each other. If two clock are physically independent then the timing analysis of circuits driven by both clocks will be misleading. A typical example is a design that has an internal clock and a clock supplied by a peripheral. The `set_clock_groups` (or `set_false_path`) commands can be used to perform timing analysis on each clock "domain" separately.

The setup and timing requirement of flip-flops with asynchronous inputs are bound to be violated at some point. Even though it's not possible to do timing analysis on asynchronous signals, it is possible to determine how often timing violations happen when signals cross clock "domains" and the consequences. This will be covered in more detail later.

Multicycles

In some cases the launching and latching clock edges can be separated by multiple clock cycles. The `set_multicycle_path` command can be used to allow a delay longer than one clock period.

Output Skew

The `set_max_skew` (Altera-specific) command constrains the maximum variation in delay between any of a group of signals (e.g. bits of a bus). This is also a simple way to specify timing constraints for source-synchronous outputs with non-critical timing constraints.

Delays

In some cases it's desirable to specify maximum or minimum delays without reference to a clock. An example would be a maximum propagation delay for a combinational logic function. The `set_max_delay` and `set_min_delay` commands can do this.