The Arrow of Time:
Following Timing Constraints in an RTL to GDSII Flow

Dwight Hill
Synopsys

The process of transforming a technology independent design representation in VHDL or Verilog into a process specific mask set is commonly referred to as an "*RTL to GDSII*" flow. In a classic flow this involves logic synthesis, followed by placement, then routing, and then buffering as distinct steps, with part or all of the process repeated in a loop until design goals are met. Current practice is to tie these functions together within a single flow. Most of the presentations and articles on this area concentrate on the algorithmic issues in logic synthesis, floorplanning, or placement. But in everyday practice the management of timing constraints forms a parallel flow, which can be almost as demanding and consume as much designer effort as the physical synthesis process.

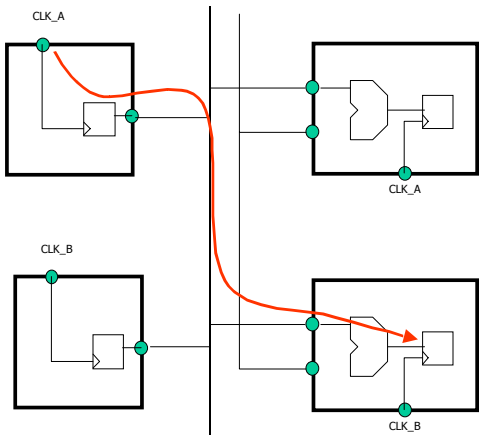
This paper serves as an introduction to the issues in supporting timing constraints through the process. As background, we first review the nature of timing constraints in current design practices: where they originate, how they evolve, and how they are represented. Secondly we consider how they must be manipulated by the tool to fit into a practical floorplan flow. Each of these steps requires its own view of the design. Finally, we summarize the characteristics of any desirable flow.

What are Timing Constraints?

Timing information can be divided into two categories, clock specifications and timing exceptions. Verilog and other RTL languages represent the logical behavior, or logical structure of a design and are often used for timing simulation, but give little information about timing exceptions. (In fact, many physical synthesis flows begin with structural Verilog, which is completely devoid of timing information.) In an ideal world only clock specifications would be needed. There would be only a few clocks, and the relationship among them would be symmetric and clear. In practice, designs with thirty or more distinct clocks (with different periods) plus potentially hundreds of derived clocks are common. Likewise an ideal circuit would have few, or no timing *exceptions*. In practice, designs tend to have a large fraction of their paths (10% is not uncommon) marked as false paths, and a similar but usually smaller number marked as multicycle paths or other exceptions.

Where do these timing exceptions originate? In the best case the set of exceptions would be obvious from the initial design specification. For example many SOC devices have common busses that act as interchange points among independent subsystems. Unless properly handled, these can appear as timing violations when a signal launched by one clock is caught by another unrelated clock. The architect generally knows that this is not a supported operation, and that the path is not exercised in the normal flow of operation. But unless properly informed, the timing engine will generate violation warnings about them. This is the first, and better source of timing exceptions: those generated proactively by system architects using knowledge of the intended system behavior.

A typical path between clock domains is illustrated below:



This path could be disabled in one of several ways, e.g.

```
 set_false_path -from [get_clocks
clk_a] -to [get_clocks clk_b]
```

or

```
 set_false_path -from
reg_a_left.out -to reg_b_right.in
```

Note that the design's intention is probably the inter-clock relationship. If the EDA tool translates this into timing exceptions directly on the registers, such information will be lost.

Unfortunately another source of exceptions is also common. These originate from the process of running timing analysis and finding violations, with a *post-hoc* analysis used to justify that the path in question is not valid. The problem with the second approach is that you can never be sure that you have a complete and correct set of timing exceptions. Each timing analysis run requires a circuit, an estimated or extracted set of loads, and a set of constraints. Early in the design process, the load estimates may have large errors in them, making the design faster on some paths and slower on others than the final tape out will observe. Pessimistic loads are create lots of "false positive" timing violations. Worse than this, even if only a small percentage of loads are optimistic but these happen to occur on true paths, timing analysis will not catch any errors. The result will be of very limited value. While this may seem to be an unlikely event in a "random" circuit, modern physical synthesis technology can actually exasperate the problem. This is because physical synthesis generally transfers delay from tightly constrained paths to unconstrained paths. Thus if a false path is incorrectly left active, it will pull resources (both CPU during the EDA tool run, and silicon area/power) from real paths. And if a true path is left underconstrained it is likely to be slowed down so much as to cause circuit failure in the final chip.

Formats for Timing Constraints

Essentially all timing constraints start out has human generated ASCI text files, usually in SDC (Synopsys Design Constraint) or some similar format. (SDC is a subset of the Tcl language accepted by PrimeTime and other timing tools). Since exception scripts are human generated they tend to be relatively compact and contain clever procedural or compacted notations for multiple constraints. For example, the command

```
 set_false_path -from [get_pins
blocka/out*] -to [get_pins
blockb/in*]
```

would be a common example. More complex example might be

```
 proc set_tc {fromblockname  toblockname}
{
     foreach inp [get_pins -filter
"direction==in &&
        type==signal" -of
$from_blockname]  {
       foreach outp [get_pins -filter
"direction==out" -of $to_blockname]  {
              set_false_path -from $inp -
to $outp;
              }
       }
      }

# now invoke the proc
set_tc blocka blockb
```

The above Tcl proc has the same effect as the original statement (more or less) but is more convenient, especially when multiple blocks must be processed because the "proc" can be reused

and does not depend on the names of the pins (only their direction).

This type of procedural format entered by the user is generally the most compact representation. However, in many flows it is not used directly. Rather, there are several types of transformations done on the constraints before tape out.

Perhaps the simplest is to eliminate the procedural aspect by executing the script. In the example above, the script is loaded and the Tcl proc "*set_tc*" is run, resulting in dozens (or hundreds) of individual false path statements executed. These may be interpreted directly by the timing engine, or stored for further processing.

The second level of processing is eliminating the wild card ("*") in object names. This is not necessarily done in the Tcl interpreter: even tools that are not based on Tcl may support this. The expansion may be done internally, by associating the constraint with many objects internally. After this expansion, the constraints are generally stored from one design object to/or through anther specific design object. The memory required for this representation is generally proportional to the number of objects involved.

In either of the above cases the intent behind the original text representation may be corrupted or be completely lost. That is, the tool may represent the constraints internally but not be able to reconstruct them for the user, except as a flat representation. Thus a few lines of SDC input can be expanded considerably.

Hierarchy Mapping

In addition to the expansion due to the text to in-memory processing, many flows involve moving constraints across the design hierarchy. These are typically at least two hierarchies the designer is concerned with. The "original" logical hierarchy, often corresponding to a behavioral RTL representation, is typically very deep. At the other end of the process, the "final" physical hierarchy corresponding to mask data (or at least to extracted routing) is usually very shallow. Thus the hierarchy takes several forms and each may have impact on the constraints. For example, if constraints are originally expressed using pins from hierarchical blocks but the design calls for those blocks to be flattened the constraints need to find a new "home" or they will be lost. Typically, this calls for algorithms that will propagate the constraints from a block pin down to a leaf cell pin. This mapping is not necessarily one-to-one. In some cases, a hierarchical block pin can propagate down to hundreds of leaf pins.

Another type of hierarchy manipulation occurs when individually designed blocks are combined into one chip, especially during System on Chip (SOC) design. The constraints associated with each block are typically written for the context of the block, often by a team in a remote location. Once the block arrives at headquarters for consolidation, it must fit onto the die. Less obviously, the timing constraints associated with the block must fit into the overall chip's timing scheme.

This process can be complex and time consuming, especially if the blocks were developed independently. One obvious problem is that each block tends to use the names of objects (pins, cells) locally, like "nand17/in2". But when processed at the chip level, they must be prefixed with the block name, e.g. "blockC/nand17/in2". More subtle problems arise from clocks, which are generally global in nature as far as the timing verification is concerned. So clocks defined within one scope may conflict with others. In both cases the most widely used solution is to modify the names of the objects with a block prefix. This tends to increase file size and decrease readability.

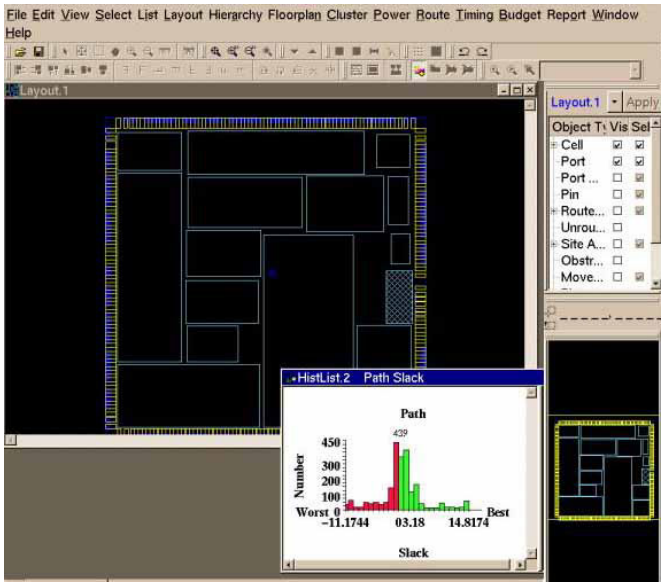
Time Budgeting

After the blocks are assembled and the constraints are made legal there is the issue of making the chip actually meet timing requirements. This can be thought of as a series of operations that identify problems (sometimes potential problems) and assign resources to them. The problems are usually slow paths. They may be due to poorly structured logic within a block, long paths of logic that run through two or more blocks, or paths that involve very long cross chip signaling without adequate buffering. The resources that

can be assigned may be obvious: larger drivers, buffering, clock-tree style synthesis. Or they may be "virtual" in that time may be apportioned away from one block and given to another, in preparation for a complete or partial physical synthesis run. This process is known as *time budgeting*.

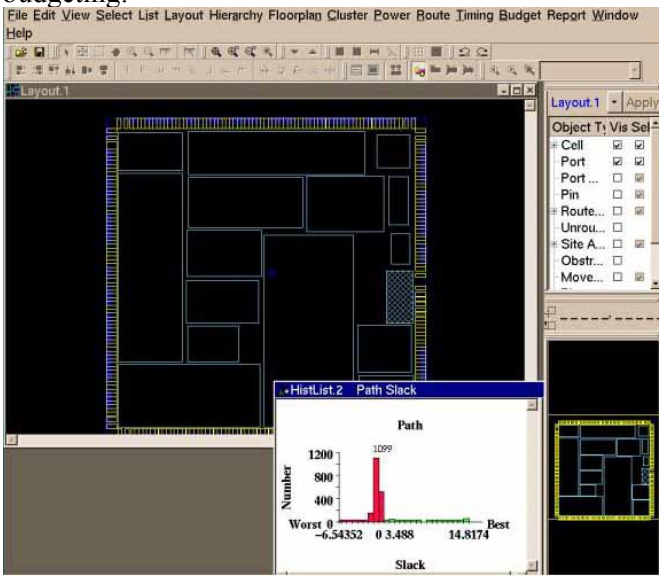
The input to time budgeting is generally a circuit early in the design process with preliminary versions of the logic and estimations of the load and delays among blocks, although time budgeting can be also used later in the process when the structure and delays are more precisely known. Time budgeting starts with the timing it is given and attempts to solve problems by "spreading the pain" out across the blocks. Thus if a path has a negative slack, the delays associated with blocks that it traverses will be reduced in the budgeting model, according to heuristics that attempt to parallel the potential for speed up in the final circuit. Similarly, paths with positive slack have their elements slowed down. Eventually the model reaches a point with zero slack and the complex process of generating timing constraints for each block can begin.

The figure below shows a typical design before time budgeting:



In this case, you can see the distribution of slacks, both positive and negative.

The figure below shows the same design after budgeting:

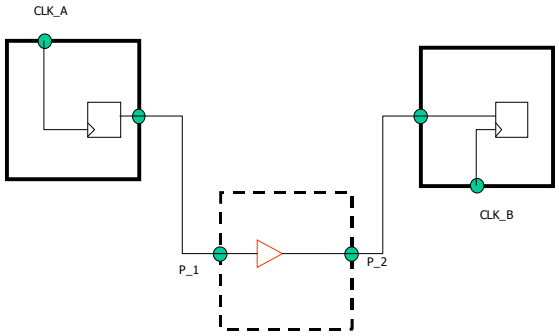


In general, time budgeting will attempt to eliminate all positive slack, but may chose to leave some small negative slack on selected paths. This is because over constraining physical synthesis may result in somewhat excessive run time, but under-constraining physical synthesis will result in the chip failing to meet timing after re-assembly.

Transforming Chip-level Constraints to Block-Level Constraints

Modeling the context of each block involves several steps and can be as complex as the overall timing analysis of the chip. The timing context of each block is a function of the surrounding logic, which can involve the entire chip. The set of clocks for each block includes not just those directly by the block but the set of clocks used by paths that enter, or even just pass through the block. Input and output characteristics of the ports of the block may reflect these other "virtual" clocks and may have independent specifications for rise time, fall time and other conditions. But the biggest source of complexity in these budgeted contexts is usually the timing exceptions. Paths within the block that have timing exceptions need to be processed, as well as paths that enter, leave, and/or pass through the block. Exceptions may be expressed as between

pins, ports, nets, clocks, or any combination thereof.



In order to process this model correctly, the clocks in the adjoining blocks must be represented in the dashed block, even though they have no physical embodiment in the block. "Virtual" clocks are used to represent them. The constraints in the dashed block might look like:

```
create_clock -name clka_virtual01
-period 7.0;
create_clock -name clkb_virtual01
-period 8.0;
set_input_delay -clock
clka_virtual01 3.25 [get_ports
p1]
set_output_delay -clock
clkb_virtual01 4.75 [get_ports
p2]
```

ECO

Many articles have discussed techniques for engineering changes (ECO's) of logic function, but there is usually little consideration to the ECO process for timing constraints. Yet timing constraints are typically as complex, and certainly as prone to error as RTL logic. Systems that flatten and expand timing constraints make it very difficult to make changes in the constraints after the floorplan is well underway. What started out as a 10K line procedural SDC file may be expanded to more than 1 million lines of text. When changes are needed in the intended chip level timing behavior, it is usually awkward (at best) to modify the expanded constraint file.

Other technologies

One distinct aspect of the time constraint processing is that it is dependent almost entirely on the complexity of the design, not on the underlying technology. Thus while other EDA problems, such as test generation might be simplified when designing with FPGA's (which tend to be 100% testable without vectors), timing analysis and constraint processing are not simplified through their use. Timing constraints are likely to continue to grow in complexity with each generation of technology.

Summary: Effective Processing of Time Constraints

The complexity and volume of timing constraints in the flow of a chip can be overwhelming. Current experience tends to show about one line of timing constraints are generated per placeable object. So for a SOC with 2 million objects (and perhaps 200 RAM's, or other blocks), one would expect an overall timing exception file of about 2 million lines by the time of sign off.

How can this situation be improved? There is no single simple answer but in general a key to effective processing is to keep constraints in the original format as long as possible, and avoid expanding the constraints as much as possible. Thus a system that starts by reading the constraints and expanding them internally is not likely to be as effective as one that can continue to accept the constraints in the user format deep into the flow. And similarly, a flow that is required to flatten, expand or enlarge them will have disadvantages over one that can output them in a more compact format.