WP331 (v1.0.2) January 18, 2008

Timing Closure 6.1i

By: Rhett Whatcott

NOTE: The material contained within this document is out of date, but it is provided as a historical reference.

The high performance of today's Xilinx devices can overcome the speed limitations of other technologies and older devices. Furthermore, designs that formerly only fit or ran at high clock frequencies in an ASIC are finding their way into Xilinx FPGAs. However, it is imperative that designers have a proven methodology for obtaining their performance objectives. This article is written specifically to address timing closure in high-performance applications.

Consider these guidelines a road map for improving performance and meeting your timing objectives. Similar to a road map, you may find detours, different and faster paths, and newer roads that will help you to achieve your goals.

© 2004-2008 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

Introduction

This updated flow includes new techniques, and updates to software flow and tools based on ISE™ 6.1i design tools and FPGA architectures.

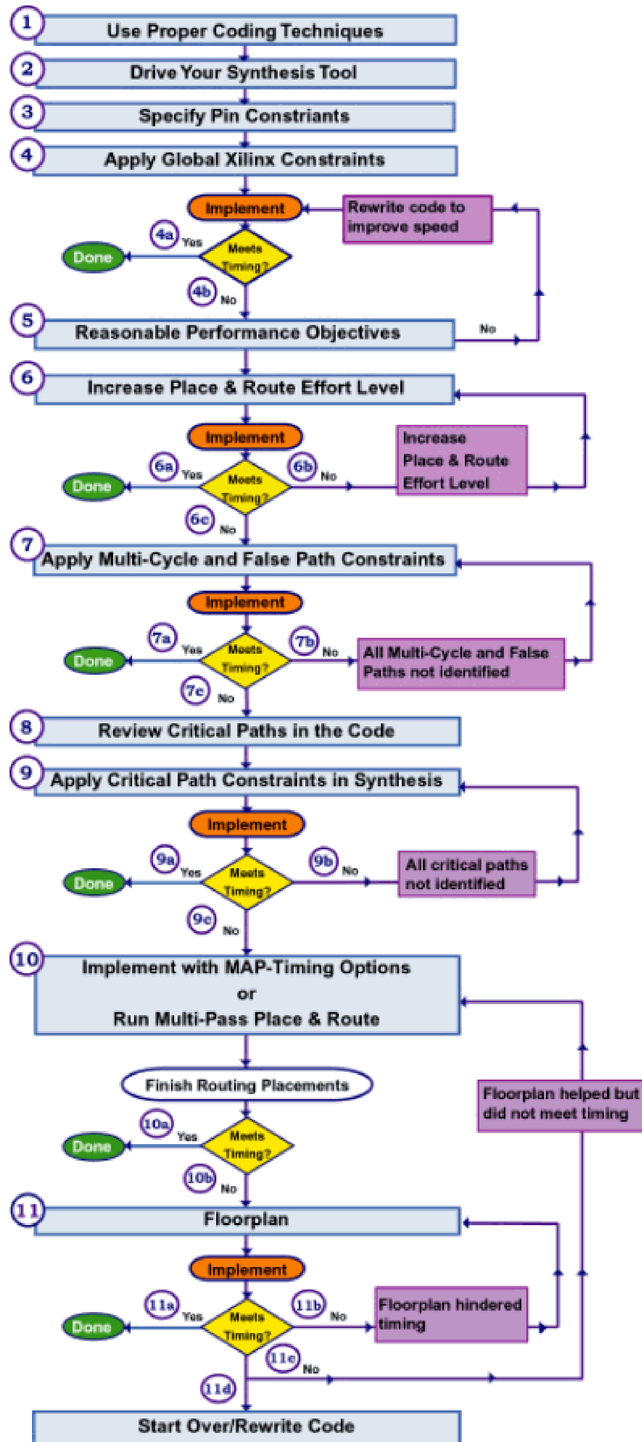

Figure 1: **Timing Closure Flowchart**

Timing Closure Flow

1. **Use proper coding techniques.** This includes many topics expanded on below, but primarily involves the implementation of synchronous design techniques, use of Xilinx specific coding, and use of cores. In this document, see the following sections:

- "Code Guidelines"
- "Synthesis"
- "Static Timing Analysis"

2. **Drive your synthesis tool.** Apply period and input/output constraints to drive optimization results from synthesis. Multi-cycle and false paths can also be applied. Maintain hierarchy to enable easier debugging in static timing analysis as well as improve your opportunities to floorplan and to implement incremental or modular design techniques. See the following sections:

- "Synthesis"
- "Static Timing Analysis"

3. **Specify wise pin constraints.** Pin constraints are often required early in the design cycle so that board development can begin. Use your knowledge of the FPGA's fabric and the design to create pin constraints that will take advantage of the FPGA architecture, design flow, and board requirements. See the following section:

- "Pin Constraints"

4. **Apply global timing constraints to the Xilinx implementation tools.** If you meet your performance objectives, you are finished. If you do not, you can later apply path-specific timing constraints. Implement the design. See the following sections:

- "Timing Constraints"
- "Static Timing Analysis"

5. **Verify that your timing objectives are reasonable.** Check the logic-level delays of your critical paths in the Post-Map Timing Report or the Post- Place-and-Route Static Timing Report. Identify how much of your timing budget is logic delay versus routing delay. Use the 60/40 rule: 60% logic (maximum), leaving 40% (or more) of your timing budget for routing delays. See the following sections:

- "Code Guidelines"
- "Static Timing Analysis"

6. **Change the implementation effort level.** A higher effort level may meet timing constraints without having to apply advanced timing constraints, use advanced tools, or change the code. Implement the design. See the following sections:

- "Implementation Options"
- "Static Timing Analysis"

7. **Apply path-specific timing constraints.** That is, apply multi-cycle, false path, and critical path constraints to help the implementation tools prioritize the placement and routing of paths. This can be an iterative process. Start by identifying the multi-cycle paths and false paths. If there are still paths that fail, constrain the critical paths with the use of a From:To constraint to give it a higher priority. Implement the design. See the following sections:

- "Timing Constraints"
- "Static Timing Analysis"

8. **Review the code.** At this point, you may find that you can save iteration time by making certain that your code is fully optimized. By using the timing analysis report generated by

Xilinx, you may be able to identify paths in your code that are not fully optimized. Implement the design. See the following sections:

- "Code Guidelines"
- "Static Timing Analysis"

9. **Apply critical path timing constraints for synthesis.** This can be an iterative process by which you identify critical paths to the synthesis tools so that it can work harder and try different algorithms to reduce the delay on critical paths. Implement the design. See the following sections:

- "Synthesis"
- "Static Timing Analysis"

10. **Use the Map -timing option or the Multi-Pass Place-and-Route tool.** The Map -timing option performs a portion of the mapping and place and route process on critical paths in the design. This can be very useful option for meeting timing constraints on a non-floorplanned design. If using the Map -timing options does not work, you can next try using Multi-Pass Place-and-Route. This tool can be very beneficial in finding a placement that will meet or come close to meeting your performance objectives. Implement the design. See the following sections:

- "Static Timing Analysis"
- "Implementation Options"

11. **Floorplan your design.** Use the Xilinx Floorplanner tool, PACE, or a third-party physical synthesis tool to meet your timing objectives. The idea behind using these tools is to specify locations on the die that the logic be placed, effectively guiding the place and route tools on placing logic. However, you can also make your timing worse since the tools cannot override your placement constraints. Implement the design. See the following sections:

- "Synthesis"
- "Static Timing Analysis"
- "Floorplanning"

Educational Services Courses

To learn more about these methods, check out these Xilinx Education Services *courses*:

Fundamentals of FPGA Design - This one-day course provides you with an introduction to designing with Xilinx FPGAs using the Xilinx ISE design tools. The course highlights the Virtex™-II device family, but it also applies to the Spartan™-3 and Virtex™ device families. For more emphasis on improving overall design performance, take the follow-up Designing for Performance course, which builds upon the basic principles covered in the Fundamentals course.

Designing For Performance - Learn how to improve design performance and manage software runtime with this two-day Fundamentals follow-up course. Topics in this intermediate level course include HDL coding techniques, advanced timing constraints, Multi Pass Place and Route (MPPR), the CORE Generator™ system, HDL simulation, and board layout.

Advanced FPGA Implementation - Advanced FPGA Implementation tackles the most sophisticated aspects of the ISE 6 tool suite. Learn to create and edit constraints for hand placing logic and creating timing constraints. Build RPMs to improve performance on critical paths. Use modular or incremental design techniques in addition to Floorplanner to implement "divide and conquer" methodologies. Take advantage of the Virtex series and Spartan™ series

clock resources with efficient clocking schemes. Optimize Post Place-and-Route design in the FPGA Editor for more efficient in-circuit testing.

Code Guidelines

(Steps 1, 5, and 8)

Only so much can be done to improve performance through software options. Most improvements in timing closure can be traced back to proper techniques in coding.

The most important condition of timing closure is the use of synchronous design techniques.

Table 1: **Synchronous design techniques and benefits**

Synchronous Design Methodology	Benefits
One clock	Reliability; faster and easier static timing analysis
Use of D-type flip-flops	Reliability; faster and easier static timing analysis
One clock edge used to clock data	Reliability; using more than one edge requires a closely controlled duty cycle
In place of multiple clocks, use clock enables	Reliability; eliminates clock skew problems; faster and easier static timing analysis
Where more than one clock is required, synchronization occurs in one hierarchical block	Minimize problem areas to one location for easier analysis
Do NOT create internally derived or divided clocks	Reliability; eliminates clock skew problems and glitching
Do not create internal asynchronous sets/resets	Reliability; eliminates glitching that may propagate erroneous data
Hierarchy broken into structural blocks defined by functionality	Replicable results that will ease the use of timing constraints; faster and easier timing analysis, debug, portability, use of advanced tools (floorplanning, use of guide files)
Leaf-level hierarchical blocks based on the path type (i.e., control, data-path, random logic, etc.)	Synthesis optimization techniques can be used based on the type of logic and the performance objectives of each block; optimization is increased as cores and Xilinx specific coding can be used now, while portability will be easier when targeting newer Xilinx technologies
Registered leaf-level outputs for each behavioral hierarchical block.	One of the most important considerations that will greatly ease the application of timing constraints over multiple iterations by preserving hierarchy during synthesis; performance; floorplanning; reducing compilation time; predictable interface timing between leaf-level blocks; easier debug, portability, and application of timing constraints; this implies that you are keeping like-logic together for better optimization; floorplanning techniques can be more easily applied

There are a few exceptions to these rules. For example, the use of the Virtex DLL or Virtex-II DCM can divide, phase shift, and multiply clocks safely. Also, double-data rate registers can

safely be used when the DLL is used to correct the input duty cycle to provide a 50% duty cycle.

Another example might be registering of inputs of each leaf-level or registering of both inputs and outputs of each leaf-level (this may be done to guarantee high performance when leaf-level blocks might be placed or floorplanned some distance away on the die). If following this guideline might break an algorithm, this can prevent outputs from being registered. In this situation it might be possible to move logic into a different block to allow this rule to be followed.

Latches

Why is the use of internal latches considered such a problem in FPGA designs? In an ASIC design they are used to improve the throughput of a path. However, in an ASIC the routing can be more carefully characterized than in a FPGA. This makes it difficult to analyze the timing through the latch and to eliminate glitches on latches that use a gated clock.

Often when latches are used, a new clock is generated (usually a gated clock – uh, oh!), which will likely glitch and introduce clock skew. It also makes timing difficult if the duty cycle is not carefully controlled.

You can remove unintended latches from your code by making certain to always cover every output condition for if-then-else and case statements in combinatorial process or always blocks. This means you must cover all possible branches and make an assignment to each output in every branch.

To do this, make a default assignment (assignment to a constant value) to each output above the if-then-else or case statement. In this way, the output will be assigned the default value and will change only in the branch where it is "set" or active.

Coding Techniques

To achieve high performance, the guidelines for synchronous design will enable you to meet your timing objectives. Now, let's consider specific coding techniques that should be used to help minimize delays between registers.

You must first understand something of the technology and approximate the amount of logic levels that you can have between registering the data. Remember that the Xilinx architecture is based on four input functions implemented in a look-up table. Use the 60/40 rule for logic and routing — allow 60% of the timing budget to be used up in logic, saving the other 40% to be used for routing.

As an example, if you want to run at 200 MHz in a Virtex-II -5 part (delay through LUT to X/Y output is 0.44 ns), you should limit the code to use no more than 6 logic levels before being registered. That amounts to a 4096-bit wide function implemented in 6 levels of logic, and it could still meet timing. This is not a likely function and may be somewhat limiting since you have to keep in mind the fan out (albeit a limited factor in Virtex-II devices) of each net. You must also consider the placement of that logic. In other words, give yourself a cushion to allow for placement and routing — you might therefore limit the function to 256 inputs, which is still a very wide function.

In most cases, your critical path will likely include state machines, arithmetic logic, or multiplexing. Below are some guidelines that should be used to reduce the delays on these paths:

Coding Guidelines

General Coding Guidelines

- Use Synchronous design techniques.
- Most synthesis tools can handle if-then-else statements and create a parallel structure. However, it is still good coding practice to follow these guidelines:
    - Use case statements rather than "if-then-else".
    - Do not nest if-then-else or case statements more than 3 deep.
        - If you must, make sure you use case statements.
    - Use parenthesis to group logic into 4 bit logic equations.
    - Use parenthesis to group arithmetic functions into 2 busses.

FSM Specific Guidelines

- Limit state machine designs to one level of hierarchy, and do not include random logic in the same block.
    - This will limit the amount of sharing of resources that the synthesis tool can perform. While sharing of resources can produce smaller logic, it can also mean longer delays (the classic trade-off).
    - Do not perform multiplexing or arithmetic logic inside the state machine, this can inhibit the performance.
- Separate the next-state decoding and output decoding into two separate processes or always blocks.
    - This will also prohibit the synthesis tool from sharing resources.
- Register the outputs of a state machine.
- Use one-hot encoding.

Xilinx Specific Guidelines

- To increase the ability to port code to new devices, use generate statements (VHDL) or ‘ifdef statements (Verilog) to create architecture specific code.
- Use Xilinx cores to decrease engineering time and money.
    - DCM, memories, multipliers, low-level functions, high-level functions.
- Add pipeline stages between logic to increase performance (throughput) at the expense of adding latency.
- Use three-state buffers to replace large multiplexers.
    - Virtex based FPGAs use dedicated AND-OR logic to implement three-state buffers.
        - Exception: Spartan™-3 FPGAs.
    - Can reduce multiplexer delays (improve performance).
    - Will reduce LUT count (improve area, decrease number of LUTs utilized).
    - Internal 3-state buffers driving the same line should all be in the same hierarchical level.
- Use one-hot inputs whenever possible.
    - Xilinx FPGAs have abundant registers, so by using this technique you may use more registers but you minimize the decoding.
    - Three-state multiplexers can be very efficient if the enable signals are one-hot encoded.
- Use SRL (Shift Register LUT) in place of registers.
    - An SRL uses a single LUT for a 16x1 shift register, replacing 16 registers.

    - Code characterization: serial-in, serial-out, addressable dynamic output, no reset capability.
- Use LUT as Distributed RAM in place of registers.
    - A Distributed RAM uses a LUT for a 16x1 RAM, replacing 16 registers and address decoding.
    - Code characterization: synchronous write, asynchronous read, no reset capability.
- Xilinx CORE Generator and Architecture Wizard allow you to optimize cores for Xilinx.
    - Use cores to reduce engineering time.
    - Use when resources are difficult to infer, difficult to design, or do not create acceptable implementation results.
- No latches in FPGA design.
    - Latches in an ASIC are used to increase throughput; in an FPGA it is harder to characterize and control the delays.
        - Results in glitching.
        - Harder to calculate timing using static timing analysis.
        - Latches often don't use the global clock, but instead use some internally derived clock.
    - To remove unintended latches, follow these guideline in your code:
        - Cover all branches of if-then-else or case statement.
        - Assign a value to each signal (or reg) in each branch.

Cores

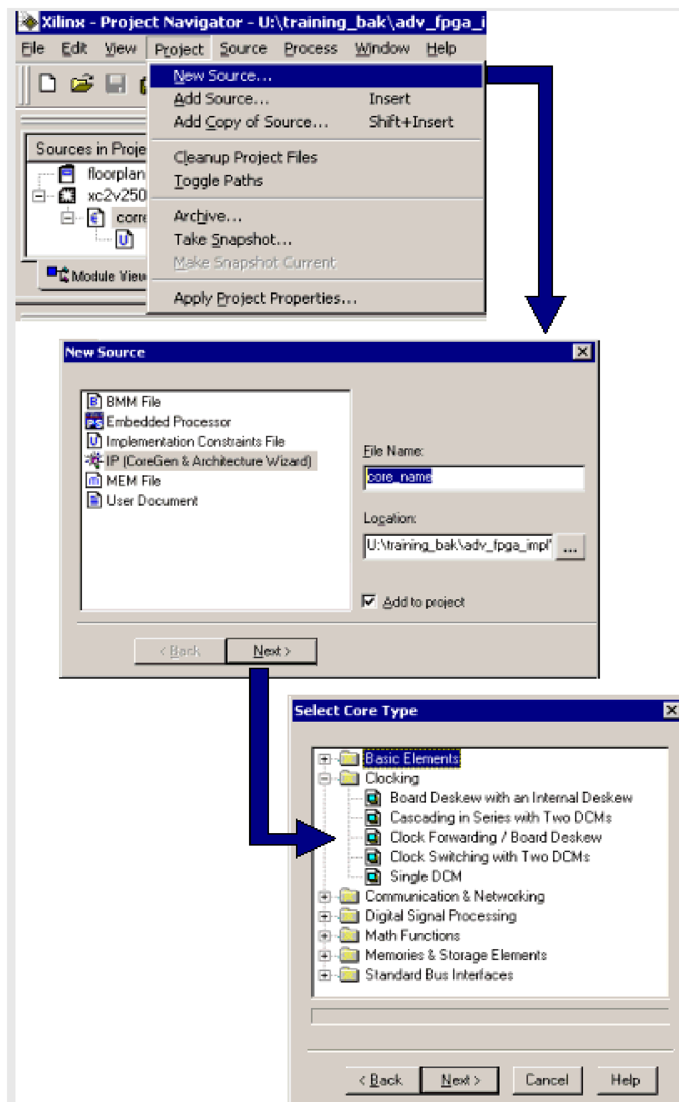

Figure 2: **Xilinx CORE Generator and Architecture Wizard**

Use the Xilinx CORE Generator and Architecture Wizard tool for higher optimization. Xilinx suggests that you keep your code portable and only instantiate cores or primitives when:

1. You cannot infer the proper resources.
2. Synthesis does not meet you timing or density requirements.
3. Use of a core will save engineering time and money.

Therefore, the use of cores is not limited just to the high-level cores. This may also require the instantiation of a multiplier that can be easily pipelined within the adder tree stages. Many basic cores are available at no cost and can be used to increase the optimization of your Xilinx device, which increases its performance. Strict adherence to the hierarchy guidelines will greatly aid the use of cores in your design, making it easy to port the code to a different architecture.

You should create cores for the following Xilinx FPGA resources:

- DCM resources
- Clock buffers
- Block RAM memories
- Multiplier blocks
- Dual-Data Rate registers
- SelectIO™ resources

The Architecture Wizard allows you to create clocking cores that can include a DCM and the required clock buffers. It also allows you to configure the DCM based on your design requirements. The configuration information is placed in your HDL code and can also be exported to a UCF file.

Synthesis

(Steps 1, 2, and 9)

Use your synthesis tools correctly and follow their guidelines to obtain the fastest performance. This will likely require that you learn several different synthesis tool directives that can be applied either in the code or through the synthesis tool's constraint editor. Whenever possible, maintain portability by entering these constraints into the synthesis tool's constraint editor.

Hierarchy Preservation

(Steps 1 and 2)

Preserve the hierarchy during synthesis. If you have followed the synchronous design guidelines from above, preserving the hierarchy will result in better performance than if you were to flatten the design. Preserving the hierarchy will also help when it comes time to perform static timing analysis and you want to compare the paths in the timing analysis report to the actual code. The Xilinx tools will also preserve hierarchical references passed to it by the synthesis tool, making it easier to find areas in the code that are preventing you from meeting your timing objectives.

Timing Constraints

(Step 2)

Apply a global period constraint for timing-driven synthesis tools (Xilinx XST, Mentor Graphics® Precision Synthesis, Synplicity® Synplify®, and Synopsys® FPGA Compiler II). These tools will try to optimize all paths in the design to meet these constraints. In most cases, you will find that over-constraining the period specification in synthesis will help you to meet timing faster. The basic trade-off is that you will spend more time during synthesis, but will reduce the time spent meeting your true timing objectives with the implementation (Place and Route) tools.

Since implementation may take several iterations and more time per iteration than synthesis takes to complete optimization, the extra time spent in synthesis is well worth the effort. In most cases, specify a period 1½-2 times faster than your true timing objective. Remember that for an FPGA, the synthesis tools are trying to estimate the routing delays and at times may not provide accurate estimates.

When you over-constrain the synthesis tool this way, do NOT forward the timing constraints (in the form of a netlist constraint file [.ncf]) from the synthesis tool to the Xilinx implementation

tools. This will result in either long run times or, more likely, errors during the place and route phase of implementation.

After attempting a few implementations and finding your critical paths, you can come back to the synthesis tool and apply path-specific timing constraints to help meet your objectives. This is an iterative step that you will perform only once your critical paths have been identified in the Xilinx implementation tools

Bottom-Up Compile

(Steps 1 and 2)

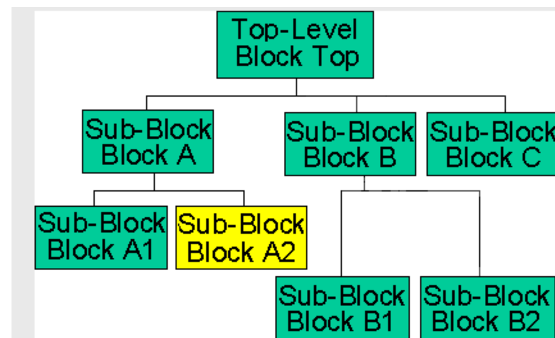

Figure 3: **Bottom-Up Compile**

As in many of these topics, to use a bottom-up compilation approach, it is vital that the synchronous design techniques previously introduced are used. Use of a bottom-up compile can be very beneficial in many ways. One benefit is to reduce compile run time, as only those blocks in the hierarchy that change need to be recompiled.

Another benefit is that each designer can implement a design and find vital statistics about their portion of the design. For example, you can find out how much area it will consume, what the critical paths are, and whether you can expect to easily meet timing. Another key benefit is that you can optimize each block based on its needs. That is, some blocks may be speed sensitive while others should be optimized for area based upon the logic type (control, data-path, random logic, etc.).

One final benefit that deserves mention, but which will not be developed here, is the use of a guide file. When a bottom-up compile approach is used, you recompile only each block that changes. Register naming (as opposed to most combinatorial logic names) is deterministic during synthesis. Therefore, a small change in a flattened design can have a ripple effect in that all the names of all the combinatorial logic change.

However, when a bottom-up compile approach is used, only the combinatorial logic within that block will change. Since using a guide file is based on naming, the changes in names will be limited, therefore enabling the use of a guide file to reduce implementation run times and keep the previously obtained "good" results. This is the beginning stages of what is known as Incremental Design techniques. Incremental Design techniques are beyond the scope of this discussion. However, it is a great methodology for maintaining good results and reducing compilation time. Incremental Design techniques are taught in the Advanced FPGA Implementation course. (For more information, visit the Xilinx Education Web site.)

Physical Synthesis

(Steps 2 and 9)

The use of physical synthesis tools is becoming a popular method by which to meet performance objectives. One of the primary objectives is to constrain related logic more closely together on the die. So, rather than expecting the place and route tools to meet timing based solely on timing constraints, you are guiding the placement of logic to help it meet the timing objectives.

This is especially important for the large devices available today. It is always suggested that you try implementing the design first without using physical synthesis tools. Generally, however, these tools can help to provide a 10-20% improvement in timing. To use physical synthesis tools, it is highly suggested that you practice all the synchronous design techniques. Again, this methodology will greatly enhance the capability of physical synthesis.

The physical synthesis tools work in a manner somewhat similar to the Floorplanner tool that is provided with all Xilinx tool sets. However, there are some key differences. In both tools you will need to provide layout information. To provide layout information, you create a rectangular space on the die to place a hierarchical block. In most situations, you will provide this information after the design has been placed and routed once so that you can identify your critical paths. Next, the physical synthesis tool applies location constraints to the blocks of logic in the critical path(s) and passes those constraints to the implementation tools.

You then implement the design with these constraints applied and find out if you have met timing. If you have not met your timing objectives, you iterate this process.

Pin Constraints

(Step 3)

Pin assignment is often an afterthought for an FPGA designer. Often it is done early in the design cycle to allow the board development to occur in tandem. Board requirements, like the placement of other devices relative to the FPGA it interfaces, are often the only factor that is considered by the designer when specifying pin constraints. However, by paying a little attention to the FPGA's fabric and the design flow, a quality pinout may be specified.

There are many issues that affect pin assignments: board layout, signal integrity, design, FPGA fabric, and so forth. Board layout and signal integrity are beyond the scope of this discussion, thus we will concentrate on taking advantage of the design flow and FPGA fabric.

Most designs have an overall data flow. We can take advantage of the flow of logic along the data path. However, other types of logic will likely exist, such as control logic, clocking, and status logic. The other types of logic may need a centralized location to make it easy to meet timing. On the other hand, some of that logic may be spread out on the die to have closer contact to the source or load that it interfaces. What we have to do is to account for the data path separate from the control logic.

The FPGA fabric is very well set up for a horizontal data flow (left to right data flow, or right to left data flow). The carry chain runs vertically, setting up the arithmetic logic to supply data in columns with the LSB at the bottom and the MSB at the top of the arithmetic logic. Likewise, block RAMs and block multipliers are stacked in columns. Three-state buffers (if used) run horizontally. Some of the FPGA families even have extra routing that runs vertically for routing control signals. Thus, the basic data flow of Xilinx FPGA devices is as shown in the Figure 4.

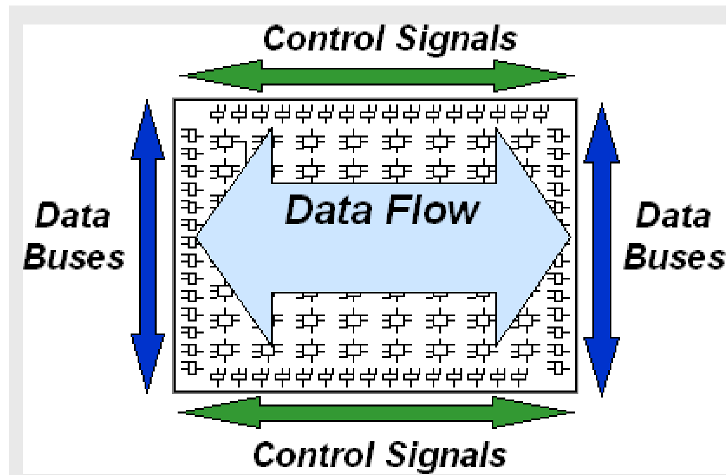

Figure 4: **Basic Data Flow of Xilinx FPGA Devices**

There are other issues to consider as well, such as specifying compatible standards in each of the eight I/O banks, using the different standards, and for high-speed serial I/O (e.g., LVDS), specifying the correct pin pairings. This is made considerably easier by using PACE. PACE stands for Pinout and Area Constraint Editor. We focus here on its use as a GUI for specifying pin constraints. PACE provides details about specifying I/O location constraints in a simple, easy to use tool.

The PACE graphic below provides a picture of the package pins. Banks are color-coded to allow you to easily drag-and-drop compatible I/Os into a bank. Dual-purpose pins and power/ground pins are easily identified. PACE also allows you to specify the I/O standard and finally to perform a design rule check to verify your pinout does not disregard FPGA banking rules.

Additionally, PACE checks for simultaneously switching output (SSO) violations. By viewing the Device Architecture window (currently behind the Package Pins window in the graphic below), you can view how the package pins relate to the IOBs on the die. By using the Package Pins window and the Device Architecture window, you can obey FPGA package rules as well as specify a pinout to take advantage of the horizontal data flow of the device.

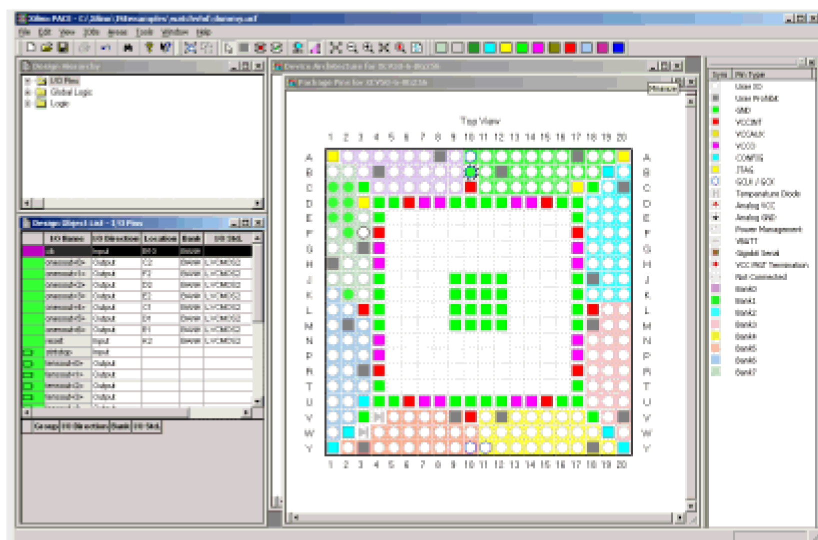

Figure 5: **Device Architecture Window and Package Pins Window**

Timing Constraints

(Steps 4 and 7)

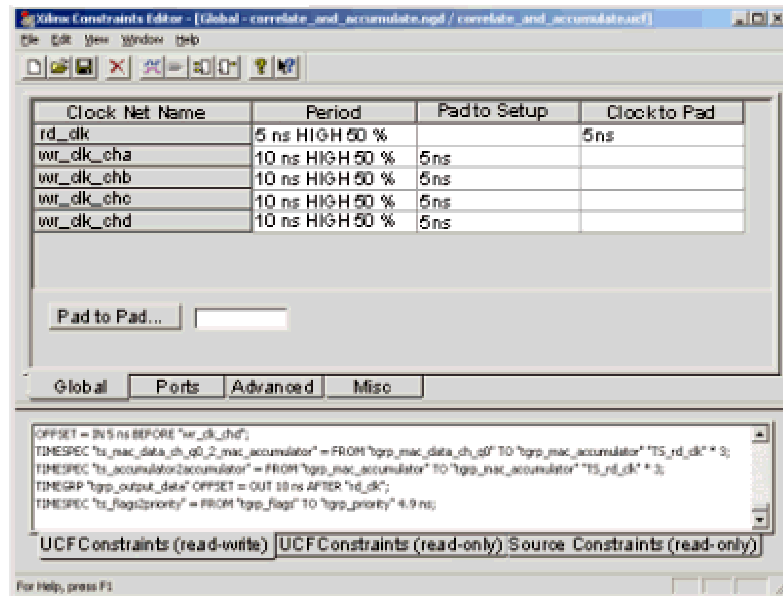

Figure 6: **Xilinx Constraints Editor**

Global Timing Constraints

(Step 4)

Timing constraints applied to the implementation tools drive placement and routing of logic on the die. Always start by applying the four global constraints for each clock: Period, Offset In, Offset Out, and Pad-to-Pad. The period constraint covers all paths between synchronous elements driven by a single clock.

The Offset In constraint covers the delay from the input pad to a synchronous element; similarly, the Offset Out constraint covers the paths from a synchronous element to the output pad. The Pad-to-Pad constraint covers all paths from an input pad to an output pad (does not cross synchronous elements).

The Period constraint has an advantage over a From:To constraint (e.g., from flip-flops to flip-flops) in that it will properly account for the duty cycle for paths that use opposite edges of the clock. The Period constraint also covers from synchronous elements to all other synchronous elements including RAMs, flip-flops and latches. The Offset In/Out constraint is advantageous over a From:To constraint (from Pad-to-FFs and from FFs-to- pads, respectively) in that it will take into account the internal clock distribution delay inside of the FPGA. While using the DLL will minimize the internal clock distribution delay, a minimal amount of skew exists that the static timing analysis tools will take into account for these constraints. Starting with ISE v6.1 toolset, you can also provide input jitter information with the Period constraint. Not only will the tools account for the input jitter, but also they will calculate the jitter that is added by the DCM and account for this in the timing report.

Path-Specific Timing Constraints

(Step 7)

Path-specific timing constraints are very useful for eliminating non-critical paths from consideration. They give the implementation software a higher degree of freedom to choose the best resources for the paths that do have tight constraints applied. Use of multi-cycle and false path constraints can, in most cases, be easily applied if you have followed the synchronous methodology as outlined earlier. Most-path specific timing constraints are based on using synchronous endpoints such as registers. Synthesis tools are deterministic in naming synchronous elements as long as the actual register name in the code does not change.

Based on these two facts, multi-cycle constraints can easily be applied and will remain applicable even after the design is re-synthesized. You can generally go forward and implement the design before concerning yourself with specifying path-specific constraints, because if you meet timing without specifying them, you may just be wasting your time in doing so.

However, if you find that the place and route run time is too long or you do not meet timing after trying a different place and route effort level, it will be worth the time and effort to identify these path-specific constraints to reduce run time and, more importantly, to meet your timing objectives.

Path specific constraints can also be specified for critical paths. By specifying a From:To constraint on a critical path (for example, TS_clock_period * 0.999), the tools will give that path a higher priority than paths covered by a global timing constraint.

Static Timing Analysis

(Step 5 and After each implementation)

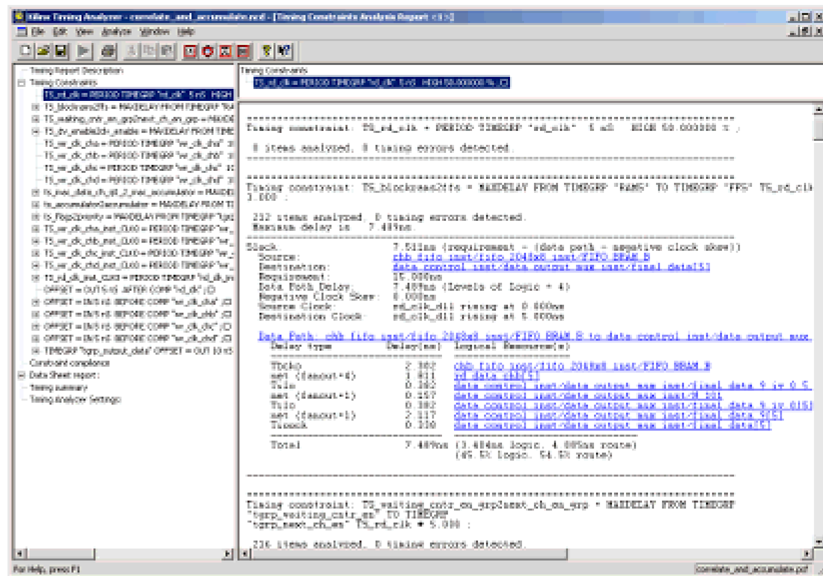

Figure 7: **Timing Analyzer**

Timing Analyzer - The Xilinx static timing analysis tool is a very productive way to find out if your design constraints are reasonable. If your performance objectives (and therefore your constraints) are not reasonable, it will mean long run times where in many situations the tools will determine that your objectives cannot be met and will therefore stop. As such, we need a proven method to find out if your constraints are reasonable.

Post-Map Static Timing Report

(Step 5)

The Post-MAP Static Timing Report (also known as the logic level timing report) is generated directly after the MAP implementation phase and before PAR. At this point, not much time has been spent in the implementation tools since most of the implementation run time is spent on the PAR phase. The Post-MAP Static Timing Report gives you a critical path analysis based on your timing constraints.

For most device families, this report provides a delay number for the minimum amount of routing needed for each path. You can ignore the minimum routing delays and concentrate on the logic delays. The logic delays will be constant throughout the final implementation. The only variable at this point is in the routing

Recall the 60/40 rule introduced earlier, and allow for 40% of the timing budget to be used in routing delay. The greater the amount of timing budget left for routing, the easier it will be to meet timing. Forty percent is a generally good guideline that identifies whether your performance objectives are reasonable given the delays on the critical paths.

So, before ever placing and routing the design, you have a very good idea of whether your timing objectives can be met. There may be situations where a path uses 70% of the timing budget for logic - this may be OK, as long as it is not true for hundreds or thousands of paths.

Post-Place-and-Route Static Timing Report

(After each implementation)

The Post-Place-and-Route Static Timing Report (also known as the post layout timing report) gives you a static timing analysis of the true delays in the placed and routed design. This is the finished product.

Remember that the only variable in the Post-Map Static Timing Report was the routing. The Post-Place-and-Route Static Timing Report will tell you whether your timing was met. If it was not met, you will know by how much. From this report, you can identify your critical paths and specify path-specific constraints in your timing-driven synthesis tool and the implementation tools.

By applying path-specific constraints in your synthesis tools, you are identifying your true critical paths to that tool and allowing it to try further optimization. By specifying a path-specific constraint in the implementation tools, you will give that path a higher priority to the tools when it performs Place-and-Route. Assuming that your constraints are reasonable, applying these path-specific constraints for both tools will give you a very good chance that you will meet timing on these paths.

For each path in the Post-Place-and-Route Static Timing Report that fails, a Timing Improvement Wizard link is provided. When you click on this link, it provides specific guidelines for decreasing the delays on that specific failing path. The information provided is not general information about improving your path, but specific information based on the delays on that path. Use the Timing Improvement Wizard to help fix large delays in your design.

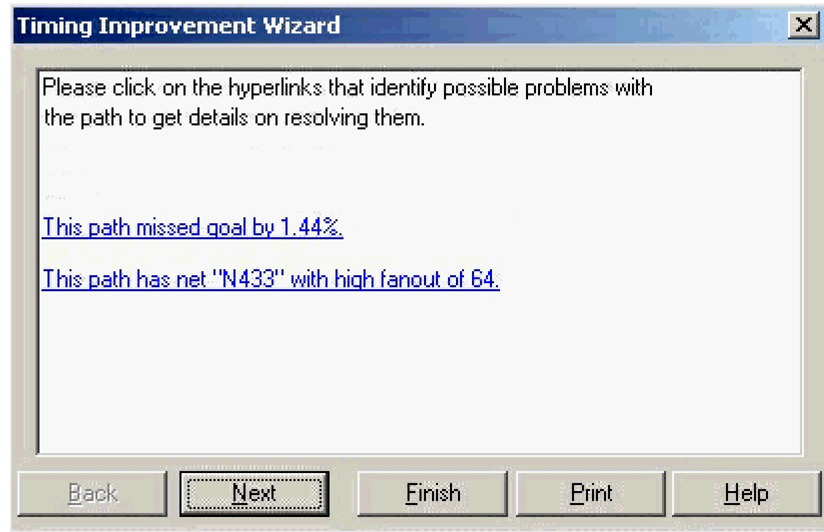

Figure 8: **Timing Improvement Wizard**

Implementation Options

(Steps 6 and 10)

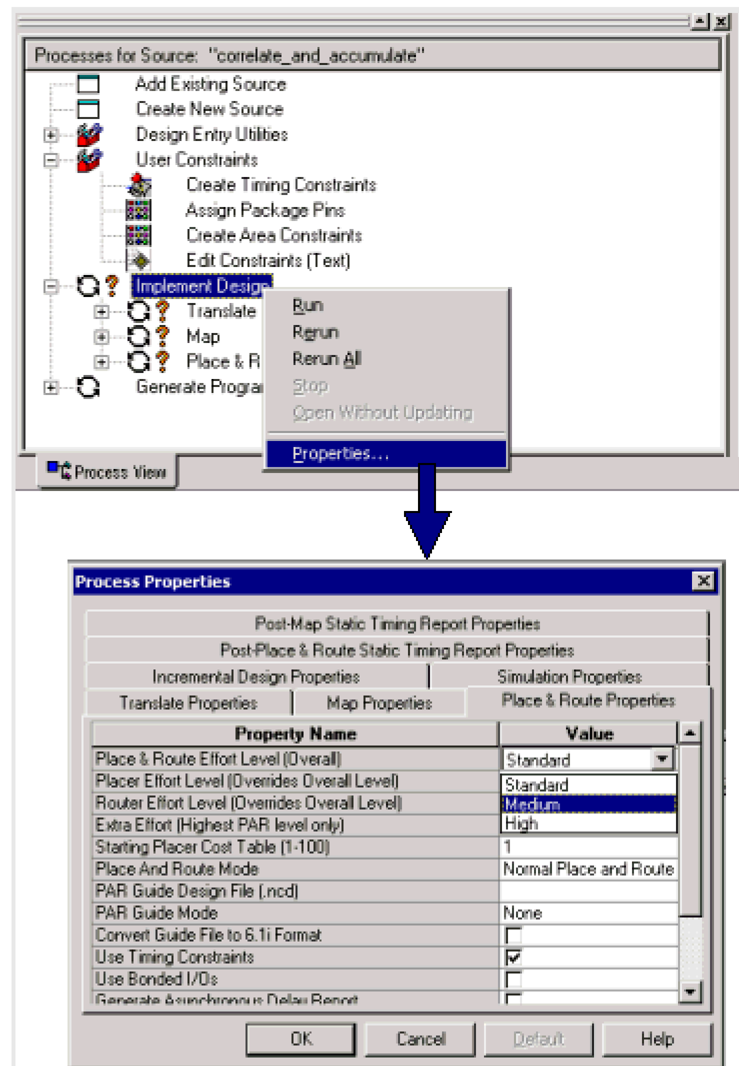

Figure 9: **Implementation Options**

Once you apply global constraints (and possibly after specifying path-specific constraints), the next step to meeting timing is to try different implementation effort levels. In proceeding through the outlined timing closure flow, if at this point you have met timing, you are finished.

If you have not met your timing objectives, the next step is to increase the place and route effort level. The default value is Standard (based on the scale Standard, Medium, High). The value of Standard will give you the fastest run time but the least effort in meeting your timing objectives. The value of High will give you the most effort at meeting your timing objectives at the expense of increased run time.

However, jumping from the default of Standard directly to High is not always the best solution. At times, an effort level of Medium can be better than High. Try effort level Standard, then Medium, then High as a final choice.

Changing the effort level will usually yield very good results without having to make changes to the code. This saves in simulation time as well as synthesis and implementation run time. However, if you have tried different effort levels without success, it is probably time to specify

multi-cycle and false path constraints and to look at ways to improve the code. Use cores or Xilinx specific coding to infer faster resources or Xilinx specific resources—this will minimize your run time for each iteration.

Advanced Implementation Options

(Step 10)

A few advanced implementation options may also help you to increase your performance: perform timing-driven packing and placement (MAP option) and extra-effort level (PAR option). The timing-driven packing and placement option uses the timing constraints to guide the packing of critical path logic into slices. Furthermore, the critical paths are placed and routed before other non-critical paths. This option has become very useful for most designs. In fact, it has become so good that for most designs using this option will provide similar results to using Multi-Pass Place-and-Route with much less run time. The extra-effort level (Normal or Continue on impossible) can be used during placement and routing to try new placement algorithms to meet your timing objectives. Try timing-driven packing with a regular PAR effort level of High first; then, try the extra-effort level starting with Normal, and try Continue on impossible last (each new level will add to the run time).

Map - Timing and Multi-Pass-Place-And-Route

(Step 10)

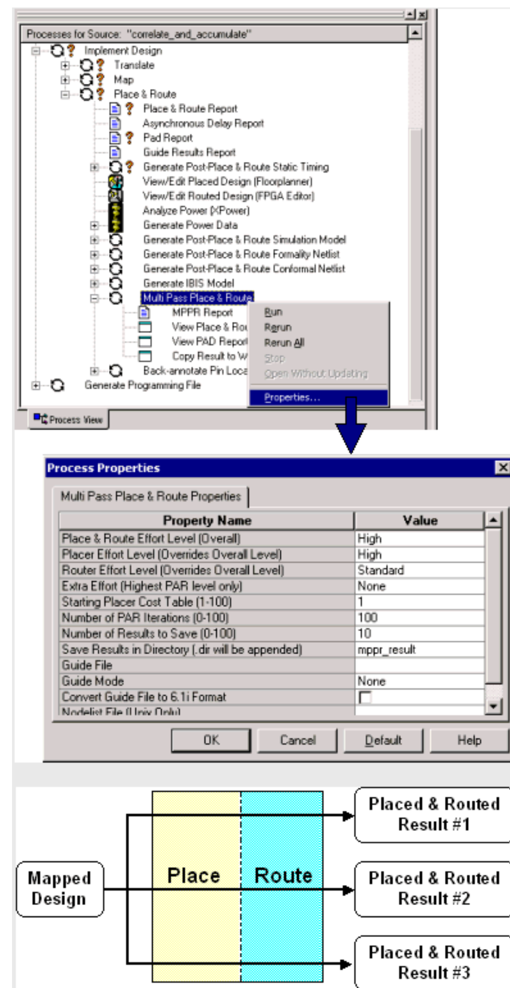

Figure 10: **Multi-Pass Place-and-Route**

Multi-Pass Place-and-Route is the part of the Xilinx tool set that fully implements the design based on a cost table (often referred to as a "seed"). 100 different cost tables can be attempted; each one will provide a fully implemented design with a different placement (and different routing), which provides different timing. Using Multi-Pass Place-and-Route is a very time-consuming task and should be used only after nearly all other options have been attempted.

One or more of these initial cost tables might provide a significant improvement in performance. The default cost table used by the implementation tools is 1. For most designs, you can expect a 15 to 20% difference in performance between the very best and the very worst cost tables. Typically, you might gain a 5% improvement over a normal place and route. However, keep in mind that cost table 1 (the default cost table for a normal place and route) may not be the most optimal "seed" for your design.

When performing MPPR, you want to find the best placement that will usually result in the best timing. The routing and timing are largely based on the placement of logic. Therefore, it is usually most beneficial to use a High effort level for placement and limit the routing effort level to Standard. In this way you can run many different placements and, once you have found the best placement, then increase the routing effort level to High to finish the routing on the best one or two.

While the quality of the routing is based on the placement, the best placement will not always produce the best timing. You must finish the router phase to find out which placement will yield the best results.

The placer phase of implementation is relatively fast compared to the router phase. For example, you might expect a Virtex-II 1000 device to complete placement within 30 minutes (often much faster) for each cost table. So, you can run all 100 cost tables over a weekend, then finish routing on the best one or two placements to find the best overall implementation. Overnight, you might be able to complete 30-40 full placements on one machine, or all 100 on three different machines. If you are using a UNIX machine, you can easily divide the work across several machines in parallel by using a script.

Floorplanning

(Step 11)

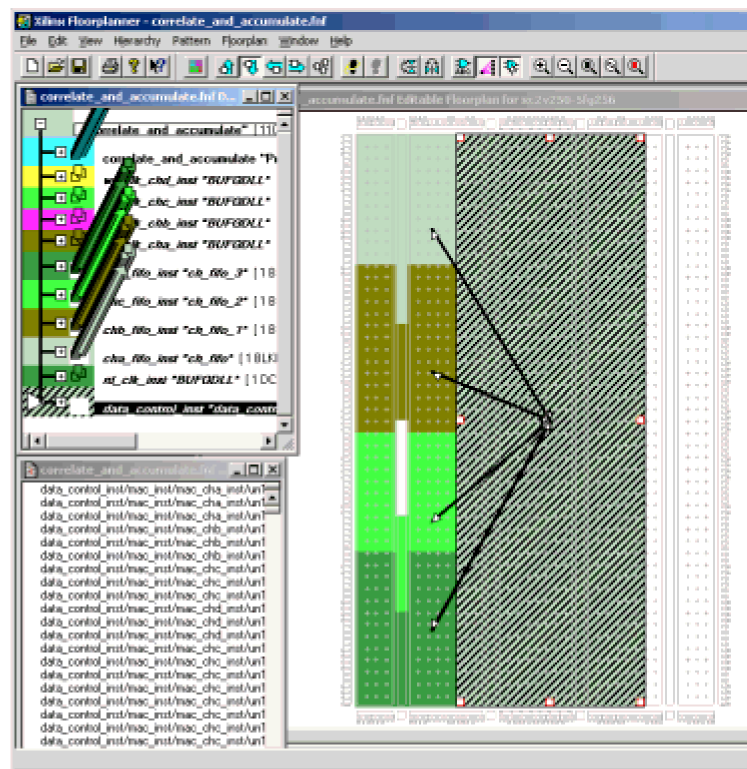

Figure 11: **Xilinx Floorplanner**

The Xilinx Floorplanner is used to specify locations on the die for logic placement. For example, it can be used to place logic in specific locations on the die, lay out locations on the die to place hierarchical blocks, create RPMs (Relationally Placed Macros), and place pins.

The Xilinx PACE (Pinout and Area Constraints Editor) tool is used to specify pin constraints and can be used to specify area constraints—that is, layout locations for hierarchical blocks on the die.

For each of these uses, it will be especially important that the design hierarchy is preserved. This is because of the hierarchy browser, which makes it convenient to locate logic and place it on the die. If you have preserved the hierarchy, the critical logic can easily be found by looking at the Post-Place-and-Route Timing Report, which will maintain the hierarchical references when the hierarchy is preserved during synthesis.

A word of warning: While the intent of floorplanning is to help guide the placement of logic (and therefore increase performance), it is possible to hurt the performance of your design, as the implementation tools cannot override your placement constraints. On the other hand, given sufficient knowledge of the architecture and software, floorplanning can be very beneficial. In fact, given enough knowledge it can be done earlier in the implementation flow than we are suggesting here, to help you to meet timing. That is, floorplanning can be done before your first implementation if you have sufficient knowledge of floorplanning, your design, and your target architecture.

Placing Logic (Floorplanner™)

You can use logic placement as a way to guide to the tools to help obtain better performance. Specifically, you might place block RAMs in specific locations so that the logic it interfaces can be placed near to that site. You should also consider the data flow within Xilinx (horizontal) to allow for the fastest possible performance. Other logic that you might floorplan is large banks of registers, distributed RAM, arithmetic logic, interleaved logic, and BUFTs. (The placement of block and distributed RAMs, interleaved logic, and BUFTs are often trouble spots for meeting timing objectives.)

To place logic on the die, simply drag the logic you wish to place from the hierarchy window onto the floorplan window to a specific location on the die. Once you have saved the floorplan information, the implementation tools will automatically attempt to use this floorplan for the next implementation.

Layout (Floorplanner and PACE)

When performing layout of the hierarchical blocks in the design, the aim is to help the implementation tools with the flow of data through the device. You must be aware of clock limitations (in Virtex-II devices), the size of each block (specifically, the length required for each block—arithmetic functions usually drive this), the number of BUFTs or internal 3-state buffers driving the same line, and the number of block RAMs in each hierarchical block you intend to floorplan. In short, you must allow for enough room for each of these.

Furthermore, if you use blocks that share 3-state buffer lines, align them such that they can all be placed on the same horizontal long line. In addition to the synchronous design techniques prescribed, it is especially useful to register both the inputs and outputs of each major block you plan to floorplan (layout). This will provide you with more freedom in placing interfacing block further apart, leaving only routing (no logic) between the two interface blocks.

You can easily begin by allowing extra room for each major hierarchical block such that you give the tools guidance in placing the logic and provide structure to the flow. If further guidance is needed, you can later add more detailed layout information for lower level blocks that exist within each major block. Note that, this limits the tools in relation to how and where logic is placed, and can mean that you hurt the performance by not allowing the tools to place related logic together.

To create a layout, select blocks of hierarchy in Floorplanner's or PACE's hierarchy window, and click on the Assign Area Constraint button; finally, draw a rectangular area on the die where you wish to place that block. As mentioned, it is usually beneficial to create an area that is slightly larger than the required area to allow the implementation tools more freedom in placing related or interface logic together.

When you have finished and saved your floorplanned layout, the implementation tools will automatically attempt to use this floorplan for the next implementation. In the past, it was considered beneficial to overlap the rectangles; however, Xilinx now recommends that you avoid overlapping areas, because the tools now actually restrict placement in overlapping areas.

About the Author: Rhett Whatcott, Senior Course Developer/Trainer Xilinx San Jose

Rhett holds a Bachelor of Science degree in Electrical and Computer Engineering from Utah State University with an emphasis in hardware design and control systems. Rhett enjoys creating training materials and teaching customers how to use Xilinx products in his capacity as a Senior Trainer/Course Developer in Customer Education Services. His work can be seen in the Designing for Performance and Advanced FPGA Implementation courses. Previously Rhett worked for Technically Speaking, Inc. as a trainer and before that as a hardware designer with L3 Communications.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
4/07/04	1.0	Initial Xilinx release.
12/10/07	1.0.1	Converted format from HTMLTechXclusive to PDF archive.
1/18/08	1.0.2	Minor Edits.

Notice of Disclaimer

The information disclosed to you hereunder (the "Information") is provided "AS-IS" with no warranty of any kind, express or implied. Xilinx does not assume any liability arising from your use of the Information. You are responsible for obtaining any rights you may require for your use of this Information. Xilinx reserves the right to make changes, at any time, to the Information without notice and at its sole discretion. Xilinx assumes no obligation to correct any errors contained in the Information or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE INFORMATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS.