XAPP418 (v1.0) August 15, 2002

Xilinx 5.1i Incremental Design Flow

Summary

This application note is directed at designers familiar with Xilinx FPGA design and constraints. Incremental Design, as a flow, can greatly decrease place and route runtimes *and* preserve design performance when making small changes to a nearly completed design. It requires that the design follow good hierarchical design methodologies, ensuring that the design is properly partitioned into separate Logic Groups. Each Logic Group is constrained, occupying its own distinct space on the Xilinx FPGA. When a design change is made to one of the Logic Groups, an Incremental Synthesis flow ensures that unchanged Logic Groups are not changed in the Synthesis output. The implementation tools then re-place and re-route the changed Logic Group (within its assigned area), while the unchanged Logic Groups are guided from a previous implementation. By guiding the unchanged Logic Groups, the performance in those Logic Groups is preserved, and place-and-route run times are decreased. This saves designers valuable time when debugging a design.

Definitions

It is important to define some of the terminology that is used with Incremental Design

- A "Logic Group" is a piece of the design that can be synthesized separately, and can be assigned to an Area Group. Typically, each Logic Group is a module in Verilog or an entity in VHDL that is instantiated in the top level of the design. See the **Identifying Logic Groups** section for more information.
- An "Area Group" is a Xilinx constraint that packs logic together during the MAP process. The AREA GROUP RANGE constraint specifies the physical location in the FPGA.
- A "small design change" is a change that only affects one Logic Group in a design. It does not drastically alter the size, nor adversely affect the timing of that Logic Group. Examples include: changes to state machines or control logic, adding registers to improve performance, and other similar types of changes.
- A "nearly completed design" is an entire design that successfully runs through the Xilinx Implementation tools. The timing requirements on the design should already be met.

Use Cases

Here are three main customer use cases:

1. The design is being debugged in the lab. The designer is finding and fixing bugs as fast as possible. The design already meets all timing requirements.
2. The design is being debugged using a simulator. The designer is finding and fixing bugs as fast as possible. The design already meets all timing requirements.
3. The design does not meet timing requirements and the designer is making small design changes to improve performance.

Identifying Logic Groups

In order for Incremental Design to significantly reduce runtime and maintain performance in unchanged portions of the design, the place and route tools must be able to look at a design in terms of the separate Logic Groups that each occupy their own space on the device. When a Logic Group is changed, the tools can completely re-place and re-route that Logic Group inside of its assigned area. Having an assigned area, which is completely open for replacement and re-routing, allows the tools to find an optimal configuration for the changed Logic Group. The placement and routing of the unchanged Logic Groups is guided from the previous implementation.

If each Logic Group is not assigned to its own area, this flow will not work very well. In this case, each Logic Group can be placed anywhere on the device and the logic from different groups may be mixed together. When a design change is made and the unchanged Logic Groups are guided from their previous placement, the changed Logic Group will be replaced, but it will have to do so by finding open spaces between the already guided logic. This makes it difficult for the tools to find an optimal placement and can increase runtime rather than decrease it. It is important to identify the Logic Groups and assign them to their own area before trying to run the Incremental Design flow.

There are three basic rules to consider when trying to divide a design into Logic Groups:

1. All logic, except IOB (Input/Output Block) logic and clock logic, should be part of a Logic Group.
2. All Logic Groups should be assigned to an Area Group.
3. Each Area Group should be assigned to its own distinct space on the Xilinx FPGA (Area Groups should not overlap).

For the reasons listed above, Logic Groups are typically defined as the Verilog modules or VHDL entities instantiated in the top level. If lower levels of hierarchy are considered as Logic Groups, the partitioning usually leaves some logic ungrouped. For example, consider a design that has a top level which instantiates A, B, and C. A has some logic in it and also instantiates D and E. If a designer defined A as a Logic Group, then the logic in D and E are covered in this group. If instead, a designer defines D and E as separate Logic Groups, then the logic in A will remain ungrouped. therefore, it is usually easier to consider A as one Logic Group. If there is no logic in A, designers can treat D and E as separate Logic Groups. In this case, there is no ungrouped logic.

Notes : Ungrouped logic is undesirable for the same reasons that the design is broken into Logic Groups in the first place.

Ungrouped logic can be placed anywhere on the device and may become an obstacle when replacing and re-routing a changed Logic Group.

Also, if ungrouped logic is changed, the placer will have to find holes in the already guided logic.

If ungrouped logic cannot be avoided, the ungrouped logic should be locked down outside of any existing Area Groups.

Design Guidelines

In addition to partitioning the design into separate Logic Groups, it is also important for the design to follow good hierarchical design methodologies. Below is a list of suggested design methodologies:

1. The design should be fully synchronous.
2. All critical paths should be within one Logic Group.
3. All IOB logic should be at the top level. Every input or output of the device should be declared in the top level as well as I/O buffers and I/O tristates. However, instantiated I/O logic in a Logic Group is acceptable.
4. Registers should be placed on all of the inputs or outputs of each Logic Group. A good design practice is to make all input signals or all output signals registered at the Logic Group boundaries. This will ensure that the critical paths inside of a Logic Group are

maintained and eliminate possible problems with logic optimization across Logic Group boundaries. Typically, it is the output signals that should be registered.

5. The top level should only contain instantiated modules or entities, IOB logic, and clock logic (DCMs, BUFGs, etc.)
6. The timing constraints on the design should be realistic.

Incremental Synthesis

In order for Xilinx Incremental Design to work, only modified Logic Groups should have updated synthesis netlist outputs. Currently synthesis tools re-synthesize the entire design, even for a small design change, by default. Therefore, Incremental Synthesis is needed to keep the output the same for any unchanged Logic Groups. A brief explanation of each is provided below.

Mentor Leonardo Spectrum

Mentor supports both a Bottom-Up and a Top-Down methodology for Incremental Synthesis, but suggests using the Bottom-Up methodology. Using the Bottom-Up methodology, separate EDIF files are created for each Logic Group and for the top level. This flow is suggested because only one script needs to be rerun when a design change is made, making it easier to manage design changes. Since the Bottom-Up methodology is suggested, it is the only method described in this section.

The first step in the Bottom-Up method is to synthesize the lower level Logic Groups. Each lower level Logic Group is synthesized in macro mode, which automatically disables I/O and clock buffer insertion. A separate EDIF file is created for each.

```
Read {a.v}
Optimize.-macro
auto-write a.edf
```

The second step is to synthesize the top level file. The previously synthesized Logic Groups are read into the database and then the top level is read. The `dont_touch` attribute is assigned to each Logic Group to prevent any optimization and the `noopt` attribute is used to prevent the Logic Groups from being written out in the top.edf.

```
Read {a.xdb b.xdb c.xdb}
Read top.v
dont_touch {a b c}
noopt {a b c }
Optimize .. –chip
```

The Xilinx Translate (ngdbuild) process will read in the top level edif and the lower level edif files to create one design file (design.ngd).

For more information please see **Appendix A: Incremental Synthesis Using Leonardo Spectrum**.

Synopsys FPGA Compiler II

Synopsys uses BLIS, Block Level Incremental Synthesis. More information on this flow is found in the FPGA Compiler II User Guide.

Synplicity Synplify/Synplify Pro

Synplify Pro always resynthesizes the entire design. This often results in different signal and component names in unchanged Logic Groups. This causes the Xilinx Incremental Design flow to assume that this is changed logic and will replace and re-route the entire design. In order to preserve unchanged Logic Groups, separate projects must be created for each Logic Group and for the top level.

The goal is to create a top level EDIF that instantiates the lower level EDIF files. A change in one EDIF file will not affect the other EDIF files. This has the advantage of faster synthesis

runtimes, because only a portion of the design will be resynthesized. Synplify attributes are used to guide the synthesis.

The first step is to create a top level EDIF that instantiates the lower level EDIF files as black boxes.

- To instantiate Logic Groups as block box, use the `syn_black_box` attribute.

The second step is to create all of the lower level EDIF files. The Logic Group EDIF files must be synthesized without inferring I/Os or clock buffers.

- To disable I/O insertion, select "Disable I/O Insertion" in the Synplify Pro GUI.
- No clock buffer will be inferenced when "Disable I/O Insertion" is turned on.

The Xilinx Translate (ngdbuild) process will read in the top level edif and the lower level EDIF files to create one design file (design.ngd).

For more information please see **Appendix B: Incremental Synthesis Using Synplify/ Synplify PRO**.

XST: Xilinx Synthesis Tool

XST supports block level incremental synthesis, within a single project. Attributes are applied to each Logic Group in the XST constraints file (.XCF) to define Logic Group boundaries. An HDL change in one of these Logic Groups will only affect that Logic Group; the rest of the Logic Groups will not be changed. For VHDL designs, detection of modified logic is done automatically. For Verilog designs, the "resynthesize" attribute must be used.

Here is an example XCF file.

```
MODEL "top"
incremental_synthesis=yes;
MODEL "A"
incremental_synthesis=yes;
MODEL "B"
incremental_synthesis=yes;
MODEL "C"
incremental_synthesis=yes;

MODEL "top" resynthesize=no;
MODEL "A" resynthesize=no;
MODEL "B" resynthesize=no;
MODEL "C" resynthesize=no;
```

For more information please see **Appendix C: Incremental Synthesis Using XST**.

Xilinx Implementation Using the Incremental Flow

This section describes the steps to go through the Xilinx Incremental Design Flow.

1. Create Area Groups for each Logic Group and assign each Area Group to a location on the device.
2. Implement the design through PAR with the Area Group constraints. This will create an initial set of guide files.
3. Make a small design change to one Logic Group.
4. Run Incremental Synthesis to update the changed Logic Group.

5. Implement the design using Incremental Guide in both MAP and PAR. The outputs from MAP and PAR in step 2 are used to guide this implementation.

Steps 3 through 5 are repeated as more design changes are made. The outputs from MAP and PAR for each new run are used to guide the next implementation (The outputs from step 5).

Notes: After running step 2, the designer may find that the shape or placement of the Area Groups has negatively affected the performance of the design or caused the design to be unroutable.

If this occurs, the Area Groups should be adjusted and step 2 should be rerun. The incremental flow should not be run until the Area Groups are optimally located.

Guidelines for Creating Area Groups

Determining good Area Group placement is the most important step in the Incremental Design flow. Bad Area Group placements can increase runtime, reduce performance and possibly create an unroutable design. Good Area Group placements can reduce runtime and improve performance.

Guidelines to follow when creating Area Groups are as follows:

1. All I/Os must be locked down.
2. Area Groups should not overlap.
3. Area Groups that communicate with each other should be placed next to each other.
4. Area Groups that communicate with I/Os should be placed near the I/Os.
5. All I/Os that communicate with an Area Group should be placed together.
6. Slice utilization inside each Area Group should be similar. One Area Group should not be 99% full and another 10% full.
7. When an Area Group contains internal TBUFs, block RAMs, or Multipliers, it can sometimes be difficult to create a Range for the Area Group that includes all of the necessary components without consuming an unnecessary amount of Slice logic. In these cases, separate ranges can be created for each. If an Area Group contains all four of these types of components, the syntax would look something like the following:

```
INST Logic_Group_A AREA_GROUP = AG_Logic_Group_A ;
AREA_GROUP "AG_Logic_Group_A" RANGE = SLICE_X0Y20:SLICE_X20Y30 ;
AREA_GROUP "AG_Logic_Group_A" RANGE = RAMB16_X0Y2:RAMB16_X0Y2 ;
AREA_GROUP "AG_Logic_Group_A" RANGE = MULT18X18_X0Y1:MULT18X18_X0Y1;
AREA_GROUP "AG_Logic_Group_A" RANGE = TBUF_X0Y0:X1Y0;
```

Notes: This Syntax is for Virtex-II™ and Virtex-II Pro™ only. Refer to the constraints guide for more information.

Separate Ranges for an Area Group can also be defined when using PACE, as is shown in the next section.

Creating Area Groups with PACE (Pinout and Area Constraints Editor)

PACE is a new tool in the Xilinx 5.1i software that is used to create pinout and area constraints. PACE takes a UCF (user constraint file) and a NGD (output from the Translate stage) as input. It outputs constraints to the UCF file that was read in or to a new UCF file. This section explains how to create an Area Group for each Logic Group and how to adjust the size and position of each Area Group.

Note: The design must be run through the Translate stage (NGDBUILD) before PACE can be opened.

Opening PACE

From Project Navigator

- Double-click on Create Area Constraints in the Processes window (Figure 1).

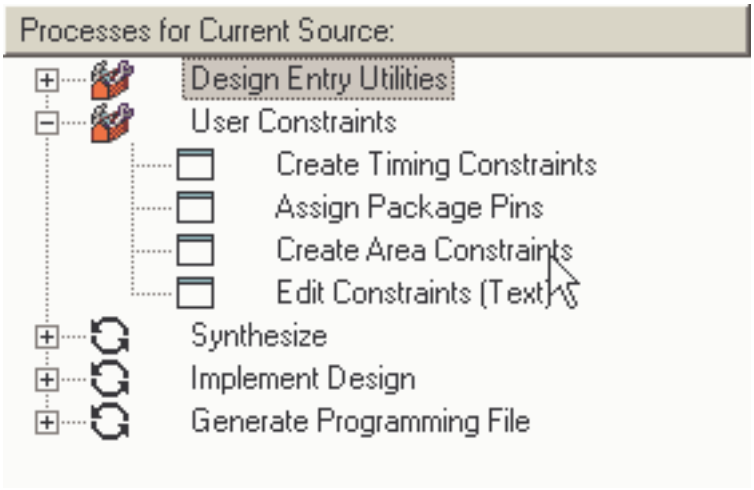

Figure 1: **Opening PACE from Project Navigator**

From DOS or UNIX

- Type "pace" at the command prompt.

From Windows

1. Click on Start > Programs > Xilinx ISE 5 > Accessories > PACE
2. In the Hierarchy Window, expand the Logic Folder. This will display the hierarchy for the design, as shown in Figure 2. Also in Figure 2, notice that the Design Object List window displays a list of components that are used in each piece of hierarchy.

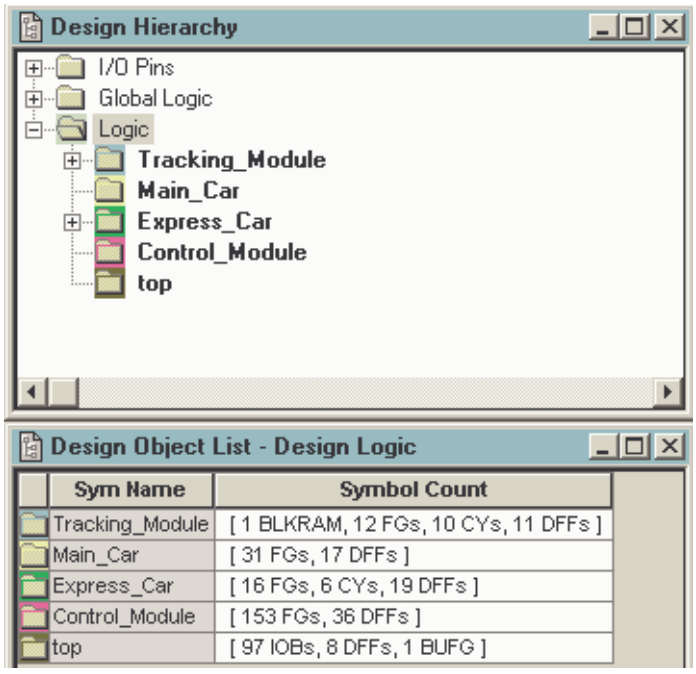

Figure 2: **Design Hierarchy**

3. Locate each Logic Group within the Hierarchy window. Typically, the upper most levels of hierarchy will be the Logic Groups (refer to the **Identifying Logic Groups** section for more information). In Figure 2, there is a top level and four Logic Groups. Each of the four Logic Groups will be assigned to an Area Group using PACE.
4. Give each Logic Group a unique color. Click on the Logic Group in the Design Hierarchy window to select it and then click on one of the colors in the toolbar (Figure 3).

Figure 3: **Color Bar**

5. Create an Area Group for each Logic Group. Highlight one of the Logic Groups in the Hierarchy window and then click on the Assign Area Constraint Mode Icon (Figure 4).

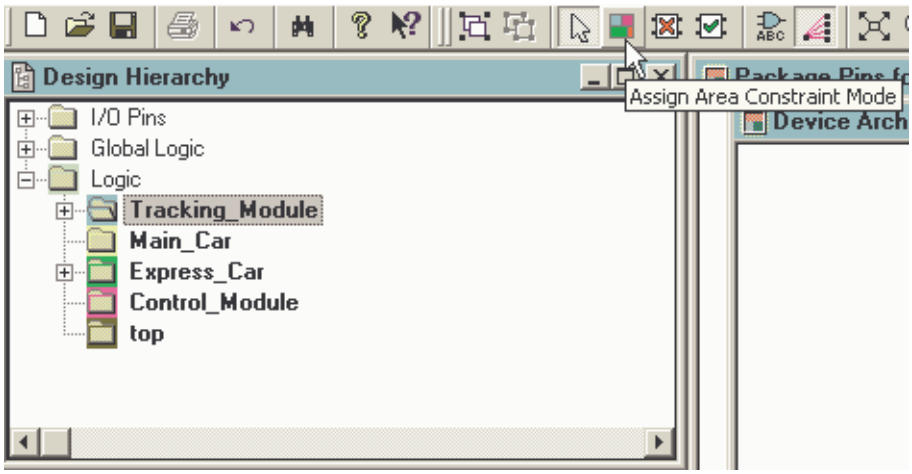

Figure 4: **Creating an Area Group Using the Assign Area Constraint Mode Icon**

6. After clicking on the Assign Area Constraint Mode Icon, draw the Area Group in the Device Architecture window by clicking with the left mouse button and holding it down until the desired size is created (Figure 5).

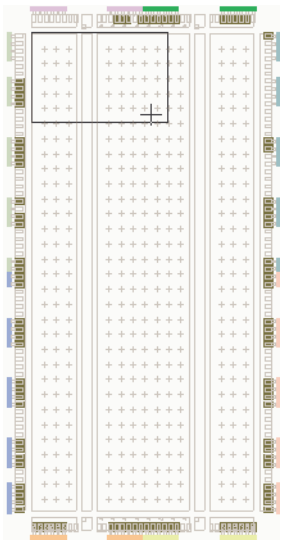

Figure 5: **Drawing an Area Group**

7. When the left mouse button is released, the Area Group will appear with the selected color (Figure 6).

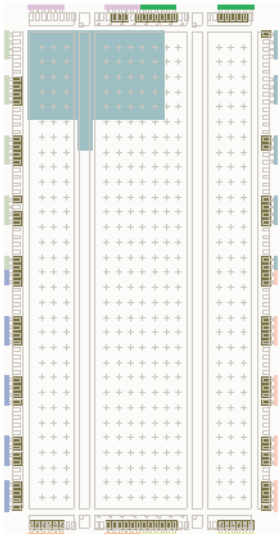

Figure 6: **Viewing the Newly Created Area Group**

Notes: In Figure 6 the Area Group is not exactly the same shape as what was drawn. This is because the Tracking_Module contains a block RAM and so PACE automatically created two separate Ranges. One range covers the CLB logic and the other range covers the block RAMs. MAP and PAR will consider these two ranges as two separate AREA GROUPS, one for the slices and one for the block RAM.

This will also happen for TBUFS and Multipliers.

If at least part of a block RAM or Multiplier is not selected when the Area Group is drawn, PACE will not automatically create a separate Range. It is important to make sure that the Area Group Range or Ranges include all of the necessary components.

8. Adjust the Area Group so that it has the desired size and move it to the desired location. Refer to the Guidelines for Creating Area Groups section. Notice that when an Area Group is selected, PACE will show which I/Os communicate with this module. It will also show which Area Groups communicate with each other.

9. Repeat steps 5-7 for each of the remaining Logic Groups. When completed, the Area Groups may look something similar to what is shown in Figure 7.

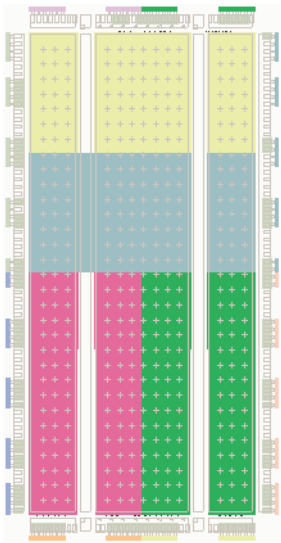

Figure 7: **Design Fully Partitioned Into Area Groups**

10. Verify that the slice utilization of each Area Group is similar. Drag the pointer over each Area Group and look at the utilization in the lower left corner of the PACE window.
11. Click on the save icon to save the Area Group constraints. This will write out the constraints to the UCF file. An example of how the constraints will look is shown below:

```
INST Tracking_Module AREA_GROUP = AG_Tracking_Module ;
INST Main_Car AREA_GROUP = AG_Main_Car ;
INST Control_Module AREA_GROUP = AG_Control_Module ;
INST Express_Car AREA_GROUP = AG_Express_Car ;
AREA_GROUP "AG_Tracking_Module" RANGE = SLICE_X0Y21:SLICE_X15Y14 ;
AREA_GROUP "AG_Tracking_Module" RANGE = RAMB16_X0Y2:RAMB16_X0Y2 ;
AREA_GROUP "AG_Main_Car" RANGE = SLICE_X0Y31:SLICE_X15Y22 ;
AREA_GROUP "AG_Control_Module" RANGE = SLICE_X0Y13:SLICE_X7Y0 ;
AREA_GROUP "AG_Express_Car" RANGE = SLICE_X8Y13:SLICE_X15Y0 ;
```

Creating Initial Guide Files

Once the Area Groups are setup, the design is run through Implementation to create an initial set of the guide files. Below is a list of steps to create the guide files.

1. Implement the design using the output from Incremental Synthesis and the UCF file with the Area Group constraints.
2. Open the MAP report and look at the Area Group Summary near the end of the report. Verify that all of the Area Group constraints were properly read in from the UCF file. Also check to make sure that none of the Area Groups are more than 100% utilized and that the utilization of each Area Group is similar.
3. Make sure that all of the timing requirements are still being met. If they are not being met, the Area Group constraints may need to be modified.

The outputs from MAP (`<design_name>_map.ngm` and `<design_name>_map.ncd`) and PAR (`<design_name>.ncd`) will guide the unchanged Logic Groups in the next implementation.

Using Incremental Guide in MAP

The Incremental Guide mode directs Map to map the unchanged Logic Groups exactly the same way that they were mapped in the previous implementation. If the unchanged Logic Groups were mapped differently, Place and Route would not be able to guide the unchanged Logic Groups. Map guiding requires the .ngm file and the .ncd file that was output from MAP in the previous implementation. PAR can re-route any signal in order to completely route the design or to meet timing.

Setting Up Incremental Guide Mode in MAP with ISE

1. Right click on MAP in the processes window.
2. Select Properties
3. In the Process Properties window:
   a. Set the Guide Mode to Incremental.
   b. Next to Use Guide Design File (.ncd), browse to and select <design_name>_map.ncd. This is the mapped NCD file from the previous implementation.

   Note: The `<design_name>_map.ngm` file will automatically be read in as long as it has the same name as the mapped .ncd file.

Setting up Incremental Guide Mode in MAP from Command Line

1. Rename the mapped .ncd and .ngm from the previous implementation to something like the following:
   `<design_name>_map_guide.ncd and <design_name>_map_guide.ngm.`

   Make sure that they have the same name or the .ngm will not be read.
2. Run MAP with the following options:
   `>map -gm incremental -gf <design_name>_map_guide.ncd`

   In the Incremental Design Flow, the same MAP options should be used for every implementation. therefore, in addition to the –gm and –gf options shown above, the options that were used in the initial implementation run should also be added to the command line.

   The MAP report will show how many slices were guided when using the Incremental Guide Mode. If no design changes are made and MAP is rerun in Incremental Guide Mode, the MAP report should show that 100% of the slices were guided. The report will look something like the following:

```
Section 8 - Guide Report
------------------------

  Guided Mapping Summary Info
  ---------------------------
  Total number of slices in guide NCDs = 128.
  Total number of guided slices = 128.
  100.0% of guide NCD slices were guided.
```

```
If a small design change has been made to one of the Logic Groups, the report
will show that less than 100% of the slices were guided. The report will look
something like the following:
Section 8 - Guide Report
------------------------
NCD slice e2c_exp_maintenance was NOT guided.
NCD slice Express_Car/N2913 was NOT guided.
NCD slice exp_floor_disp_0_OBUF was NOT guided.

  Guided Mapping Summary Info
  ---------------------------
  Total number of slices in guide NCDs = 128.
  Total number of guided slices = 125.
```

```
97.7 % of guide NCD slices were guided.
```

Using Incremental Guide in Place and Route

Incremental guide is used in par. There are three guide modes in PAR.

1. Exact — PAR keeps the placement and routing of all unchanged components and signals. This allows very little flexibility in placing and routing the changed components and signals. It is very good for guiding cores like the PCI core.
2. Leverage — PAR starts with the placement and routing of all unchanged components and signals. It can move the existing components and signals when placing and routing the new components and signals. Since it is looking at the entire design, runtimes can be long.
3. Incremental — PAR keeps the placement and routing of the unchanged Logic Groups, while the placement and routing of the changed Logic Groups is not kept. Since each Logic Group is assigned to its own location, PAR can preserve the placement and routing of the unchanged Logic Groups, while having the freedom to completely re-place and re-route the changed Logic Groups. This keeps the performance of the unchanged design while improving runtime since only one Logic Group has to be placed and routed.

Setting Up Incremental Guide Mode in PAR with ISE

1. Right click on PAR in the processes window.
2. Select Properties.
3. In the Process Properties window:
   a. Next to Guide File, browse to and select <design_name>.ncd. This is the placed and routed NCD file from the previous implementation.
   b. Set the Guide Mode to Incremental

Setting Up Incremental Guide Mode in PAR from Command Line

1. Rename the placed and routed .ncd from the previous implementation to something like the following:
   `<design_name>_par_guide.ncd.`
2. Run PAR with the following options:
   `>par -gm incremental -gf <design_name>_par_guide.ncd`

   Notes: In the Incremental Design Flow, the same PAR options should be used for every implementation. Therefore, in addition to the –gm and –gf options shown above, the options that were used in the initial implementation run should also be added to the command line.

The Place and Route report file has guide information. See below for an example. In the example, a design change was made in the AG_express_car Area Group:

```
Xilinx Place and Route Guide Results File
=========================================

Guide Summary Report:

Design Totals:
  Component Area Groups Placed:              4 out of     5   80%

  Components:
    Name matched:                          226 out of   227   99%
    Total guided:                          211 out of   226   93%

  Signals:
    LOGIC0/LOGIC1 nets ignored:              2 out of   545
    Name matched:                          366 out of   545   67%
    Total guided:                          311 out of   366   84%
```

```
Guide file:
"C:\incremental_5_1\Incremental_Design_Implementation\Implementation\increm
ental
_design_lab\top.ncd"        Guide mode: "incremental"

  Area Group:  AG_Tracking_Module.RAMB16
  Components:
    Name matched:                              1 out of     1  100%
    Total guided:                              1 out of     1  100%

  Area Group:  AG_Control_Module.SLICE
  Components:
    Name matched:                             81 out of    81  100%
    Total guided:                             81 out of    81  100%

  Area Group:  AG_Tracking_Module.SLICE
  Components:
    Name matched:                              7 out of     7  100%
    Total guided:                              7 out of     7  100%

  Area Group:  AG_Main_Car.SLICE
  Components:
    Name matched:                             24 out of    24  100%
    Total guided:                             24 out of    24  100%

  Area Group:  AG_Express_Car.SLICE
  Components:
    Name matched:                             15 out of    16   93%
    Total guided:                              0 out of    15    0%

  Signals:
    LOGIC0/LOGIC1 nets ignored:               12 out of   381
    Name matched:                            366 out of   381   96%
    Total guided:                            311 out of   366   84%
```

Description of the Design Totals Section:

- Component Area Groups Placed — Indicates how many Area Groups were guided. In the Incremental Flow, all but one of the Area Groups should typically be guided. In the above example, four out of the five Area Groups were guided.
- Name matched — Indicates how many component names or signal names in the guide file matched the names in the new design.
- Total Guided — Indicates how many of the matched components or signals were guided. Matched components and signals are not guided if they are part of a changed Logic Group.
- LOGIC0/LOGIC1 nets ignored — None of the power and ground signals are guided.

Description of the Guide File section:

- Name matched — Indicates how many component names inside each Area Group were matched.
- Total guided — Indicates how many of the matched components were guided.

  Notes: Inside each Area Group, all or none of the matched components should be guided.

  There are two AREA GROUPS for Tracking Module. One for slices and one for block RAMS.

Conclusion

The Xilinx Incremental Design Flow is used to improve runtime while keeping the performance of the unchanged Logic Groups. This is accomplished by assigning each Logic Group to an Area Group and assigning the Area Group to its own location on the device. A design change will only affect one Logic Group. Place and Route will keep the placement and routing of the unchanged Logic Groups while re-placing and routing the changed Logic Groups within their assigned Area.

Appendix A: Incremental Synthesis Using Leonardo Spectrum

Leonardo Spectrum supports both a Bottom-Up and a Top-Down methodology for Incremental Synthesis, but suggests using the Bottom-Up methodology. A description of how to use each method and example TCL scripts for each are provided below.

Bottom-Up Methodology

The Bottom-Up methodology requires that all Logic Groups and the top level be synthesized separately and that separate EDIF netlists be created for each Logic Group and for the top level. This method is recommended because only the synthesis script for the changed Logic Group needs to be rerun when a small design change is made. The steps to use this method are provided below.

Notes: The example TCL scripts below assume a design with three Logic Groups, a,b,c and a top level, top.

For more information on Logic Groups, refer to the **Identifying Logic Groups** section.

1. Synthesize each Logic Group and write out a separate EDIF file for each. When synthesizing each Logic Group, it is important to use macro mode, which will automatically disable I/O and clock buffer insertion. An example TCL script for Logic Group “a” is provided below:

```
#Clear database and libraries of any previously existing designs
clean_all

#Set the technology environment for a Virtex-II
set part 2V80fg256
set process 5
set wire_table xcv2-80-5_avg
load_library xcv2

#Read in the design file or files for this Logic Group
read -technology "xcv2"  { a.v }

#Set the timing constraints
set input2register 9
set register2output 14
set_clock -name .work.a.INTERFACE.clk -clock_cycle "10.000000"
set_clock -name .work.a.INTERFACE.clk -pulse_width "5.000000"

#Optimize the Logic Group in macro mode to prevent I/O or clock buffer
insertion
optimize .work.a.INTERFACE -target xcv2 -macro -area -effort standard -
hierarchy auto

#Optimize for timing
optimize_timing .work.a.INTERFACE

#Disable NCF creation
set novendor_constraint_file TRUE

#Write out the Logic Group as a binary XDB database and EDIF netlist
```

```
auto_write a.edf
```

Notes: The above script synthesizes Logic Group “a."

A separate script is needed for Logic Groups “b” and “c."

2. Synthesize the top level. Each of the previously optimized Logic Groups are read in and the DONT TOUCH and NOOPT attributes are applied to each. The DONT TOUCH attribute is used to prevent the previously optimized Logic Groups from being reoptimized. The NOOPT attribute is used to prevent the previously optimized Logic Groups from being written out inside the top level EDIF. An example TCL script is provided below:

```
#Clear database and libraries of any previously existing designs
clean_all

#Set the technology environment for a Virtex-II and enable register mapping
to the I/Os
set part 2V80fg256
set process 5
set wire_table xcv2-80-5_avg
set virtex_map_iob_registers TRUE
load_library xcv2

#Load previously optimized Logic Groups
read -technology "xcv2" { a.xdb }
read -technology "xcv2" { b.xdb }
read -technology "xcv2" { c.xdb }

#Read in and elaborate the top level
read -technology "xcv2"  { top.v }

#DONT_TOUCH attribute prevents each Logic Group from being reoptimized
DONT_TOUCH .work.a.INTERFACE
DONT_TOUCH .work.b.INTERFACE
DONT_TOUCH .work.c.INTERFACE

#NOOPT attribute prevents each Logic Group from being written into the top
level edif
NOOPT .work.a.INTERFACE
NOOPT .work.b.INTERFACE
NOOPT .work.c.INTERFACE

#Set timing constraints
set input2register 10
set register2output 15
set_clock -name .work.top.INTERFACE.clk -clock_cycle "10.000000"
set_clock -name .work.top.INTERFACE.clk -pulse_width "5.000000"

#Optimize top level and preserve hierarchy
optimize .work.top.INTERFACE -target xcv2 -chip -area -effort standard -
hierarchy preserve
optimize_timing .work.top.INTERFACE

#Generate Reports
report_area area.txt -cell_usage
report_delay delay.txt -num_paths 1 -longest_path -clock_frequency

#Enable NCF creation
set novendor_constraint_file FALSE

#Write out the top level as a binary XDB database and EDIF netlist
auto_write top.edf
```

3. The design is now ready to be run through the Xilinx Implementation Tools. When a design change is made, only the script for the changed Logic Group needs to be rerun and thus only one EDIF file will be changed.

   Notes: It is also possible for Leonardo to combine the entire design into one top level EDIF file. To do this, remove the NOOPT attributes.

   Without the NOOPT attribute, the previously optimized Logic Groups will be written out into the top level EDIF. This is not recommended, as both the script for the changed Logic Group and the script for the top level must be rerun when a design change is made, but it can be done if it is required to only have one EDIF file.

Top-Down Preserving Hierarchy Methodology

Leonardo Spectrum Level 3 provides the necessary hierarchy control and optimization capabilities necessary to accommodate the Top-Down preserving hierarchy methodology. Using this flow, one EDIF file will be created. One script is used to run an Initial Synthesis and a second script is used for Incremental Synthesis passes.

Initial Synthesis Using Top-Down Preserving Hierarchy

This section contains an example script for running the Initial Synthesis pass for the Top-Down preserving hierarchy method. When running this pass, it is important to set bubble_tristates to FALSE and to set no_boundary_optimization to TRUE. These settings will keep the hierarchy “pure” and unchanged. It is also important to preserve hiearchy when synthesizing. Below is an example TCL script:

```
#Clear database and libraries of any previously existing designs
clean_all

#Set bubble_tristates FALSE to ensure that the tristates will not moved
across hierachy boundary.
#Users must place all tristate I/O at the top level to use a block-based flow
set bubble_tristates FALSE

#Set no_boundary_optimization to prevent constant propagation and other
boundary
#effect optimizations. This may cause degradation in QoR if the design makes
use
of constants across hierarchical boundaries
set no_boundary_optimization TRUE

#Set the technology environment for a Virtex-II and enable register mapping
to the I/Os
set part 2V80fg256
set process 5
set wire_table xcv2-80-5_avg
set virtex_map_iob_registers TRUE
load_library xcv2

#Read the entire design with top.v as the top level module
read -technology "xcv2"  {
   a.v
   b.v
   c.v
   top.v}

#Set timing constraints
set input2register 10
set register2output 10
set_clock -port -name .work.top.INTERFACE.clk -clock_cycle "10.000000"
set_clock -port -name .work.top.INTERFACE.clk -pulse_width "5.000000"

#Optimize the design and preserve hierarchy
```

```
optimize .work.top.INTERFACE -target xcv2 -chip -area -effort standard -
hierarchy preserve
optimize_timing .work.top.INTERFACE

#Generate reports
report_area area.txt -cell_usage
report_delay delay.txt -num_paths 1 -longest_path -clock_frequency

#Write out entire design as a binary XDB database and EDIF netlist
auto_write top.edf
```

After running the above script, the design is ready for an initial implementation pass. When small design changes are made to one of the Logic Groups, the incremental script in the next section must be used to update the EDIF.

Incremental Synthesis Using Top-Down Preserving Hierarchy

When a change is made to a Logic Group, the commands for synthesizing the design are slightly different in this flow. The key to a successful incremental synthesis is to ensure that only the modified Logic Group is modified in the EDIF netlist. To achieve this, Leonardo has the ability to reload and re-optimize a changed Logic Group, while leaving the other Logic Groups unchanged. An example TCL script is provided below:

```
#Clear database and libraries of any previously existing designs
clean_all

#Set bubble_tristates FALSE to ensure that the tristates will not moved
across hierachy boundary.
#Users must place all tristate I/O at the top level to use a block-based flow
set bubble_tristates FALSE

#Set no_boundary_optimization to prevent constant propagation and other
boundary
#effect optimizations. This may cause degradation in QoR if the design makes
use
#of constants across hierarchical boundaries
set no_boundary_optimization TRUE

#Set the technology environment for a Virtex-II and enable register mapping
to the I/Os
set part 2V80fg256
set process 5
set wire_table xcv2-80-5_avg
set virtex_map_iob_registers TRUE
load_library xcv2

#Reload the previously optimized design
read -technology "xcv2"  { TOP.xdb }

#Read in the modified Logic Group
read -technology "xcv2"  { b.v }

#Set Timing Constraints
set input2register 10
set register2output 10
set_clock -name .work.top.INTERFACE.clk -clock_cycle "10.000000"
set_clock -name .work.top.INTERFACE.clk -pulse_width "5.000000"

#Optimize the modified Logic Group in macro mode
optimize .work.b.INTERFACE -target xcv2 -macro -delay -effort standard -
hierarchy preserve

#Set top as the present design
```

```
present_design .work.top.INTERFACE

#Optimize the timing of the modified Logic Group
optimize_timing .work.b.INTERFACE

#Generate Reports
report_area area.txt -cell_usage
report_delay delay.txt -num_paths 1 -longest_path -clock_frequency

#Write out the entire design as a binary XDB database and EDIF netlist
auto_write top.edf
```

The new EDIF is now ready to be run through the Incremental Implementation Flow. Each time a Logic Group is modifed, this script is used to update the EDIF file.

Appendix B: Incremental Synthesis Using Synplify/ Synplify PRO

When using Synplify Pro for Incremental Synthesis, a separate project is created for each Logic Group and for the Top Level. When changes are made to a Logic Group, the project for that Logic Group is used to generate a new EDIF file, while the EDIF files for the top level and for the unchanged logic groups will remain the same. This will allow the design to be used with the Incremental Design Flow.

Creating an EDIF for the Top Level

The first step is to create a top level EDIF that instantiates the lower level EDIF files as black boxes. Below is an example of the steps used to create the top level EDIF file.

1. Create a new Project File named TOP.
2. Click on Impl Options to select the target device
3. Add the library file for the target device. For example, if the design is coded in Verilog and a Virtex-II device is being targeted, add virtex2.v. The library files are located in \lib\xilinx in the directory where Synplify is installed.
4. Add the top level file.
    a. In the Top Level, “/* synthesis syn_black_box */” must be applied to each instantiated Logic Group. This will tell Synplify to treat each instantiation as a black box.
    b. "/* synthesis syn_isclock = 1 */” should be applied to the clock ports of instantiated Logic Groups. This will instruct Synplify to infer a BUFG if one has not already been assigned to the signal that drives this port.
5. Press Run.
    a. At this stage a top level EDIF file should be created. The EDIF will contain all of the I/O and clock logic, as well as black box instantiations of all of the Logic Groups in the design.
6. Save the Project.

Creating an EDIF for each Logic Group

The steps below describe how to create an EDIF for a Logic Group.

1. Create a new Project File with the name of the Logic Group.
2. Click on Impl Options
    a. Select the target device.
    b. Check the option to Disable I/O insertion. This will also disable clock buffer insertion.
3. Add the library file for the target device.

4. Add the file or files for this Logic Group.
5. Press Run.
6. Save the Project.

   Notes: The above steps must be followed for each Logic Group in the design.

   After EDIF files have been created for each Logic, all of the EDIF files can be copied to an implementation directory where Xilinx Incremental Design Flow can be run.

   When a design change is made, reopen the project for the changed Logic Group and recreate the EDIF file. Then copy the new EDIF file into the implementation directory and rerun the Incremental Implementation Flow.

Appendix C: Incremental Synthesis Using XST

XST supports block level incremental synthesis. An HDL change in one of these Logic Groups will only affect that Logic Group; the rest of the Logic Groups will not be changed. The unmodified portions of the design will still be parsed, but new netlists (.NGC files) will not be written.

Logic Groups are defined are defined using the incremental_synthesis attribute. Attributes to define Logic Group boundaries are entered in the XST constraints file (.XCF) or within the HDL source itself. The top level does not require this attribute. An NGC file will be created for each Logic Group and for the top level. If a module/entity is instantiated within a design more than once and is defined as the top of a Logic Group, a unique .NGC file will be created for each instance.

When a logic change is made to any module/entity within one of the Logic Groups, only that Logic Group should be reoptimized. For VHDL designs, XST automatically detects any changes to the source HDL files and resynthesizes only the Logic Groups containing modified modules/entities. For Verilog designs, the "resynthesize" attribute is required for XST to be made aware of logic changes.

You will see evidence of Incremental Synthesis in the XST log file in a number of locations. First, when the Incremental Synthesis attributes are parsed, you will see:

```
Reading constraint file C:\design\top.xcf.
    Set property "INCREMENTAL_SYNTHESIS = yes" for unit <a>.
    Set property "INCREMENTAL_SYNTHESIS = yes" for unit <b>.
    Set property "INCREMENTAL_SYNTHESIS = yes" for unit <c>.
XCF parsing done.
```

During the initial synthesis run, you will see that each Logic Group is optimized and multiple .NGC files are created:

```
=========================================================================
*                        Low Level Synthesis                            *
=========================================================================
Optimizing unit <top> ...
Optimizing unit <a> ...
Optimizing unit <b> ...
Optimizing unit <c> ...
…
=========================================================================
*                           Final Report                                *
=========================================================================
Final Results
Top Level Output File Name         : top
Output File Name                   : a.ngc
Output File Name                   : b.ngc
Output File Name                   : c.ngc
…
```

```
Finally, during the incremental pass, you will see that only the modified
Logic Group is reoptimized:

=========================================================================
*                      Low Level Synthesis                              *
=========================================================================
Incremental synthesis: Unit <a> is up to date ...
Incremental synthesis: Unit <b> is up to date ...
Incremental synthesis: Unit <top> is up to date ...
Optimizing unit <b> ...
…


Here is an example XCF file. Consult the XST User Guide for more details
about XST Constraint File syntax.

# Use the "incremental_synthesis" attribute to denote Logic Groups
MODEL "a"
incremental_synthesis=yes;
MODEL "b"
incremental_synthesis=yes;
MODEL "c"
incremental_synthesis=yes;

# The "resynthesize" attribute is only required for Verilog designs
MODEL "top" resynthesize=no;
MODEL "a" resynthesize=no;
MODEL "b" resynthesize=yes;
MODEL "c" resynthesize=no;
# End .XCF file
```

Notes: If you have previously synthesized your design without the incremental synthesis attributes, or if you have changed the structure of the Logic Groups of the design, you must remove the existing .NGC files before you can synthesize again. This is easily done within ISE by selecting Project > Cleanup Project Files.

XST needs to have an initial set of .NGC files that build the current proper hierarchy before an incremental synthesis run can be performed.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
08/15/00	1.0	Initial Xilinx release.