Development System Reference Guide

About This Manual

The *Development System Reference Guide* contains information about the command line software programs in the Xilinx Development System. Most chapters are organized as follows:

- A brief summary of program functions
- A syntax statement
- A description of the input files used and the output files generated by the program
- A listing of the commands, options, or parameters used by the program
- Examples of how you can use the program

For an overview of the Xilinx Development System describing how these programs are used in the design flow, see the "Design Flow" chapter.

Manual Contents

The *Development System Reference Guide* provides detailed information about converting, implementing, and verifying designs with the Xilinx command line tools. Check the program chapters for information on what program works with each family of Field Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD). Following is a brief overview of the contents and organization of the *Development System Reference Guide*:

Note For information on timing constraints, UCF files, and PCF files, see the *Constraints Guide*.

- Chapter 1, "Introduction" —This chapter describes some basics that are common to the different Xilinx Development System modules.
- Chapter 2, "Design Flow"—This chapter describes the basic design processes: design entry, synthesis, implementation, and verification.
- Chapter 3, "Modular Design"—This chapter provides an overview of Modular Design and describes how to run the Modular Design flow.
- Chapter 4, "PARTGen"—PARTGen allows you to obtain information about installed devices and families.
- Chapter 5, "NGDBuild"—NGDBuild performs all of the steps necessary to read a netlist file in XNF or EDIF format and create an NGD (Native Generic Database) file describing the logical design reduced to Xilinx primitives.
- Chapter 6, "Logical Design Rule Check"—The logical Design Rule Check (DRC) comprises a series of tests run to verify the logical design described by the Native Generic Database (NGD) file.
- Chapter 7, "MAP"—MAP maps the logic defined by an NGD file into FPGA elements such as CLBs, IOBs, and TBUFs.
- Chapter 8, "Physical Design Rule Check"—The physical Design Rule Check (DRC) comprises a series of tests run to discover physical errors in your design.
- Chapter 9, "PAR"—PAR places and routes FPGA designs.

- Chapter 10, "PIN2UCF"—PIN2UCF generates pin-locking constraints in a UCF file by reading a a placed NCD file for FPGAs or GYD file for CPLDs.
- Chapter 11, "TRACE"—Timing Reporter and Circuit Evaluator (TRACE) performs static timing analysis of the physical design based on input timing constraints.
- Chapter 12, "Speedprint"— Speedprint lists block delays for a specified device and its speed grades.
- Chapter 13, "BitGen"—BitGen creates a configuration bitstream for an FPGA design.
- Chapter 14, "PROMGen" —PROMGen converts a configuration bitstream (BIT) file into a file that can be downloaded to a PROM. PROMGen also combines multiple BIT files for use in a daisy chain of FPGA devices.
- Chapter 15, "IBISWriter"—IBISWriter creates a list of pins used by the design, the signals inside the device that connect those pins, and the IBIS buffer model that applies to the IOB connected to the pins.
- Chapter 16, "NGDAnno"—NGDAnno annotates timing information found in the physical NCD design file back to the logical NGD file.
- Chapter 17, "NGD2EDIF"—NGD2EDIF converts an NGD file to an EDIF file for use in simulation.
- Chapter 18, "NGD2VER"—NGD2VER converts an NGD file to a Verilog HDL file for use in simulation.
- Chapter 19, "NGD2VHDL"—NGD2VHDL converts an NGD file to a VHDL file for use in simulation.
- Chapter 20, "XFLOW"—XFLOW automates the running of Xilinx implementation and simulation flows.

- Appendix A, "Xilinx Development System Files"—This appendix gives an alphabetic listing of the files used by the Xilinx Development System.
- Appendix B, "EDIF2NGD, XNF2NGD, and NGDBuild" —This appendix describes the netlist readers (EDIF2NGD and XNF2NGD) and how they interact with NGDBuild.

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this Web site. You can also directly access these resources using the provided URLs.

Resource	**Description/URL**
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at http://support.xilinx.com/support/searchtd.htm
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://support.xilinx.com/partinfo/databook.htm
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://support.xilinx.com/xcell/xcell.htm
Technical Tips	Latest news, design tips, and patch information for the Xilinx design environment http://support.xilinx.com/support/techsup/journals/index.htm

Conventions

This manual uses the following conventions. An example illustrates most conventions.

Typographical

The following conventions are used for all documents.

- `Courier font` indicates messages, prompts, and program files that the system displays.

  ```
  speed grade: - 100
  ```

- **`Courier bold`** indicates literal commands that you enter in a syntactical statement. However, braces “{ }” in Courier bold are not literal and square brackets “[ ]” in Courier bold are literal only in the case of bus specifications, such as bus [7:0].

  ```
      rpt_del_net=
  ```

  `Courier bold` also indicates commands that you select from a menu.

  File → **Open**

- *Italic font* denotes the following items.

  - ♦ Variables in a syntax statement for which you must supply values

    ```
        edif2ngd design_name
    ```

  - ♦ References to other manuals

    See the *Development System Reference Guide* for more information.

- ♦ Emphasis in text

  If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.

- Square brackets “[ ]” indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  edif2ngd [*option_name*] *design_name*

- Braces “{ }” enclose a list of items from which you must choose one or more.

  lowpwr ={**on**|**off**}

- A vertical bar “ | ” separates items in a list of choices.

  lowpwr ={**on**|**off**}

- A vertical ellipsis indicates repetitive material that has been omitted.

```
IOB #1: Name = QOUT'
IOB #2: Name = CLKIN'
.
.
.
```

- A horizontal ellipsis “…” indicates that an item can be repeated one or more times.

  `allow block` *block_name loc1 loc2 … locn;*

Online Document

The following conventions are used for online documents.

- Blue text indicates cross-references within a book. Red text indicates cross-references to other books. Click the colored text to jump to the specified cross-reference.
- Blue, underlined text indicates a Web site. Click the link to open the specified Web site. You must have a Web browser and internet connection to use this feature.

Contents

Chapter 9 PAR

Chapter 1

Introduction

This chapter describes the basics of the different Xilinx Development System command line programs. The chapter contains the following sections:

- "Command Line Program Overview"
- "Command Line Syntax"
- "Command Line Options"
- "Invoking Command Line Programs"
- "Reading NCD Files with NCDRead"

Command Line Program Overview

Xilinx command line programs allow you to implement and verify your design. The following table lists which programs you can use for each step in the design flow. For detailed information, see the "Design Flow" chapter.

Table 1-1 Command Line Programs in the Design Flow

Design Flow Step	Command LIne Program
Design Implementation	NGDBuild, MAP, PAR, BitGen
Timing Simulation (Design Verification)	NGDAnno, NGD2EDIF, NGD2VER, NGD2VHDL
Static Timing Analysis (Design Verification)	TRACE
Back Annotation (Design Verification)	NGDAnno, NGD2EDIF, NGD2VER, NGD2VHDL

Each program has multiple options, which allow you to control how a program is executed. For example, you can set options to change output file names, to set a part number for your design, or to specify certain files to read in when executing the program.

You can run these programs in the standard design flow or use special options to run the programs in a Modular Design flow, as described in the "Modular Design" chapter.

Note The command line programs described in this manual underlie many of the Xilinx Graphical User Interfaces (GUIs). The GUIs can be used in conjunction with the command line programs. For information on the GUIs, see the online Help provided with each tool.

Command Line Syntax

Command line syntax always begins with the command line program name. The program name is followed by any options and then file names. Use the following rules when specifying command line options:

- Enter options in any order.
- Precede options with a hyphen (–) and separate them with spaces. In some cases, you can precede options by a plus sign (+).
- Be consistent with upper and lower case.
- When an option requires a parameter, separate the parameter from the option by spaces or tabs.
  - ♦ Correct: **par -l 5**
  - ♦ Incorrect: **par -l5**
- When specifying options that can be specified multiple times, precede the parameter with the option letter.
  - ♦ Correct: **–l xilinxun -l synopsys**
  - ♦ Incorrect: **–l xilinxun synopsys**
- Enter parameters that are bound to a particular option *after* the option.
  - ♦ Correct: **–f** *command_file*
  - ♦ Incorrect: *command_file* **–f**

Use the following rules when specifying file names:

- Enter file names in the order specified in the chapter that describes the program.
  - Correct: **par input.ncd output.ncd freq.pcf**
  - Incorrect: **par input.ncd freq.pcf output.ncd**
- Use lower case for all file extensions (for example, .ncd).

Command Line Options

The following options are common to many of the command line programs in the Xilinx Development System.

–f (Execute Commands File)

For any Xilinx Development System program, you can store command line program options and file names in a command file. You can then execute the arguments by entering the program name with the –f option followed by the name of the command file. This is useful if you frequently execute the same arguments each time you execute a program or if the command line command becomes too long.

You can use the file in the following ways:

- To supply all the command options and file names for the program, as in the following example:

  ```
  par –f command_file
  ```

  command_file is the name of the file that contains the command options and file names.

- To insert certain command options and file names within the command line, as in the following example:

  ```
  par –i 33 –f placeoptions –s 4 –f routeoptions
  design_i.ncd design_o.ncd
  ```

  placeoptions is the name of a file containing placement command parameters.

  routeoptions is the name of a file containing routing command parameters.

You create the command file in ASCII format. Use the following rules when creating the command file:

- Separate program options and file names with spaces.
- Precede comments with the pound sign (#).
- Put new lines or tabs anywhere white space is allowed on the UNIX or DOS command line.
- Put all arguments on the same line, one argument per line, or a combination of these.
- All carriage returns and other non-printable characters are treated as spaces and ignored.
- No line length limitation exists within the file.

Following is an example of a command file.

```
#command line options for par for design mine.ncd
-a  -n 10
-w
-l 5
-s 2 #will save the two best results
/home/yourname/designs/xilinx/mine.ncd
#directory for output designs
/home/yourname/designs/xilinx/output.dir
#use timing constraints file
/home/yourname/designs/xilinx/mine.pcf
```

–p (Part Number)

You can use the –p option with the EDIF2NGD, XNF2NGD, NGDBuild, MAP, and XFLOW programs to specify the part into which your design will be implemented. You can specify a part number at the following different points in the design flow:

- In the input netlist (does not require the –p option)
- In a Netlist Constraints File (NCF) (does not require the –p option)
- With the –p option when you run a netlist reader (EDIF2NGD or XNF2NGD)
- In a User Constraints File (UCF) (does not require the –p option)

- With the –p option when you run NGDBuild

  By the time you run NGDBuild, you must have already specified a device architecture.

- With the –p option when you run MAP

  When you run MAP, an architecture, device, and package must be specified, either on the MAP command line or earlier in the design flow. If you do not specify a speed, MAP selects a default speed. You can only run MAP using a part number from the architecture you specified when you ran NGDBuild.

Note Part numbers specified in a later step of the design flow override a part number specified in an earlier step. For example, a part specified when you run MAP overrides a part specified in the input netlist.

A complete Xilinx part number consists of the following elements:

- Architecture (for example, xc4000ex)
- Device (for example, xc4028ex)
- Package (for example, pq208)
- Speed (for example, -3)

The following table lists ways to specify a part on the command line.

Table 1-2 Part Number Examples

Specification	**Examples**
Architecture only	4000ex x4000ex xc4000ex
Device only	4028ex x4028ex xc4028ex
DevicePackage	4028exhq240 x4028exhq240 xc4028exhq240
Device–Package	4028ex-hq240 x4028ex-hq240 xc4028ex-hq240
DevicePackage–Speed	4028exhq240-3 x4028exhq240-3 xc4028exhq240-3
Device–Package–Speed	4028ex-3-hq240 x4028ex-3-hq240 xc4028ex-3-hq240
Device–Speed	4028ex-3-hq240 x4028ex-3-hq240 xc4028ex-3-hq240
Device–Speed–Package	4028ex-3-hq240 x4028ex-3-hq240 xc4028ex-3-hq240
Device–SpeedPackage	4028ex-3hq240 x4028ex-3hq240 xc4028ex-3hq240

Note Speedprint allows you to specify a speed grade. If you don't specify a speed grade, Speedprint reports the default speed grade for the device you are targeting. See the "–s (Speed Grade)" section of the "Speedprint" chapter for details.

–h (Help)

When you enter a program name followed by
–help or **–h**, a message displays that lists all the available options and their parameters as well as the legal file types for use with the program. The message also explains each of the options.

Following are descriptions for the symbols used in the help message:

Symbol	Description
[ ]	Encloses items that are optional
{ }	Encloses items that may be repeated
< >	Encloses a variable name or number for which you must substitute information
,	Indicates a range for an integer variable
–	Indicates the start of an option name
+	Indicates the start of an option name
:	Binds a variable name to a range
|	Logical OR to indicate a choice of one out of many items. The OR operator may only separate logical groups or literal keywords.
( )	Encloses a logical grouping for a choice between subformats

Following are examples of syntax used for file names:

- <*infile*[**.ncd**]> indicates that the .ncd extension is optional but that the extension must be .ncd.
- <*infile*<**.edn**>> indicates that the .edn extension is optional and is appended only if there is no other extension in the file name.

For architecture-specific programs, such as BitGen, you can enter the following to get a verbose help message for the specified architecture:

```
program_name -h architecture_name
```

Note On the UNIX command line, you can redirect the help message to a file to read later or to print out by entering the following:

```
program_name -h >& filename
```

Invoking Command Line Programs

You start Xilinx Development System command line programs by entering a command at the UNIX™ or DOS™ command line. See the chapters in this book for the appropriate syntax.

In addition, Xilinx offers the XFLOW program, which allows you to automate the running of several programs at one time. See the "XFLOW" chapter for more information.

Reading NCD Files with NCDRead

A Native Circuit Description (NCD) file contains a physical description of your design in terms of the components in the target architecture. NCDRead enables you to quickly generate an ASCII (text) file based on the data found in one or more NCD files.

To start NCDRead from the UNIX or DOS command line, type the following.

```
ncdread [–o outfile_name] filename1.ncd
{filename2.ncd ...}
```

Note Standard output goes to your screen if you do not use the –o option to write the output to a file.

Chapter 2

Design Flow

This chapter describes the process for creating, implementing, verifying, and downloading designs for FPGA and CPLD devices. For a complete description of FPGAs and CPLDs, refer to *The Programmable Logic Data Book*.

This chapter contains the following sections:

Design Flow Overview

The standard design flow consists of the following steps:

- Design Entry and Synthesis—In this step of the design flow, you create your design using a Xilinx-supported schematic editor, a Hardware Description Language (HDL) for text-based entry, or both. If you use an HDL for text-based entry, you must synthesize the HDL file into an EDIF or XNF file or, if you are using the Xilinx Synthesis Technology (XST) GUI, into an NGC file.
- Design Implementation—By implementing to a specific Xilinx architecture, you convert the logical design file format, such as EDIF, that you created in the design entry or synthesis stage into a physical file format. The physical information is contained in the Native Circuit Description (NCD) file for FPGAs and the VM6 file for CPLDs. Then you create a bitstream file from these files and optionally program a PROM or EPROM for subsequent programming of your Xilinx device.
- Design Verification—Using a gate-level simulator or cable, you ensure that your design meets your timing requirements and functions properly. See the *iMPACT User Guide* for information about Xilinx download cables and demonstration boards.

Note In addition to the standard design flow described in this section, you can break your design into modules for team-based design and run the Modular Design flow. See the “Modular Design” chapter for details.

The following figure shows the Xilinx design flow.

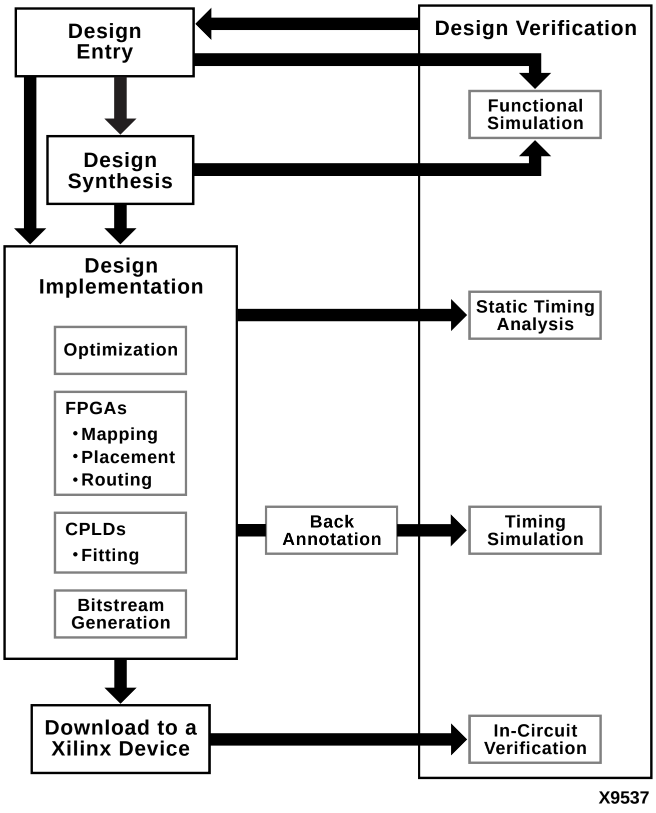


Figure 2-1 Xilinx Design Flow Overview

The full design flow is an iterative process of entering, implementing, and verifying your design until it is correct and complete. The Xilinx Development System allows quick design iterations through the design flow cycle. Because Xilinx devices permit unlimited reprogramming, you do not need to discard devices when debugging your design in circuit.

The following figure shows the Xilinx software flow chart for FPGA designs.

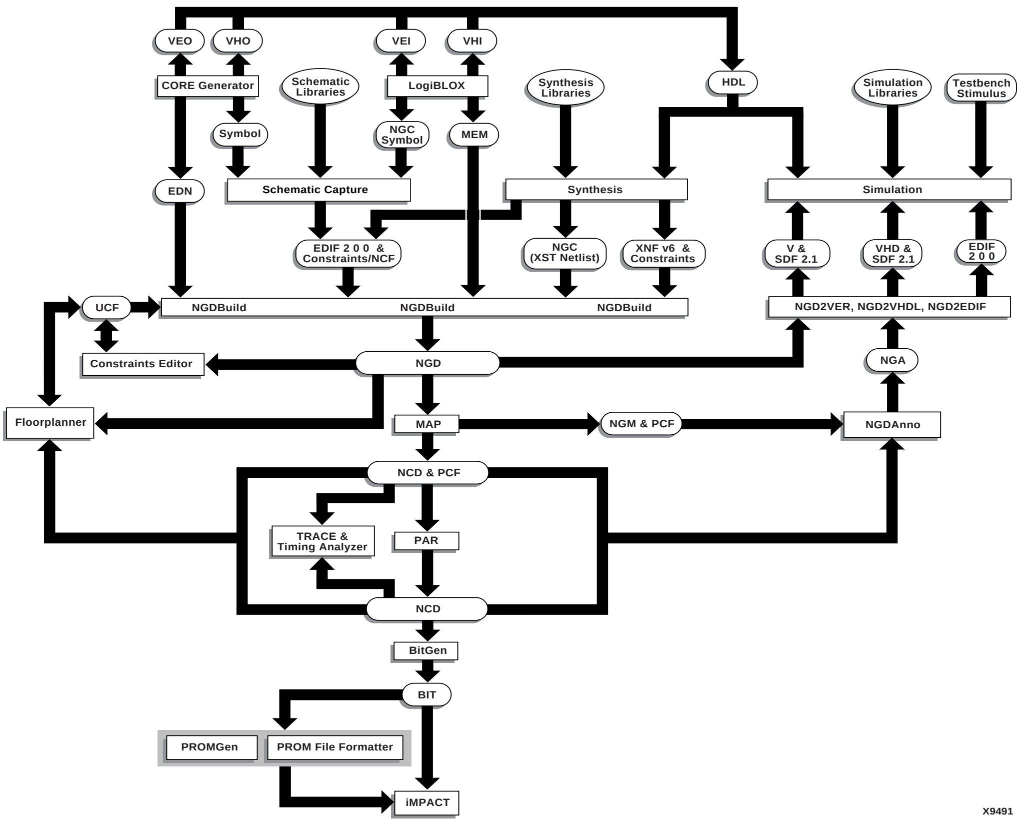


Figure 2-2 Xilinx Software Design Flow (FPGAs)

The following figure shows the Xilinx software flow chart for CPLD designs.

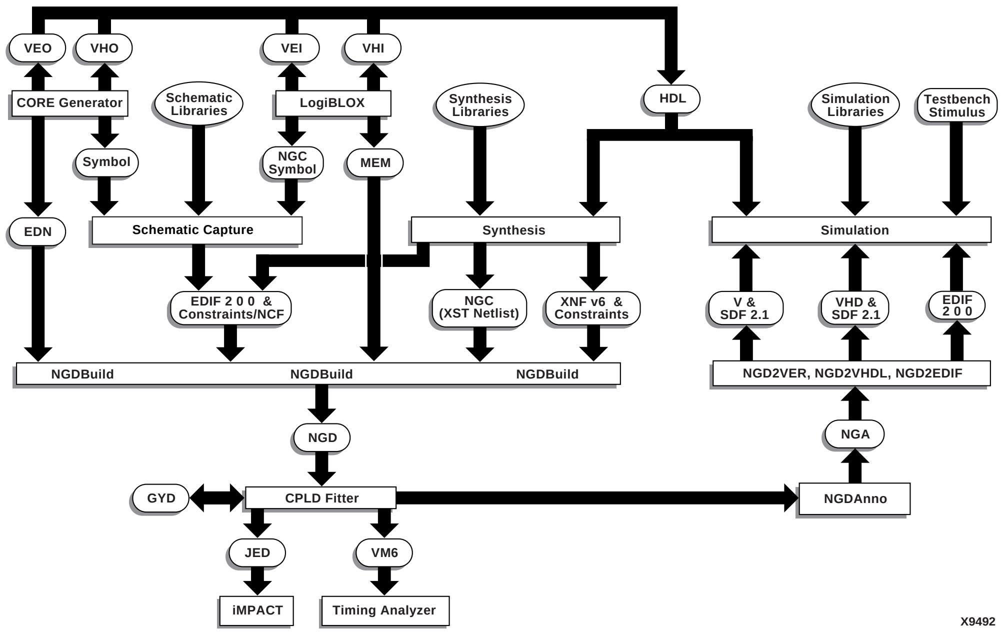


Figure 2-3 Xilinx Software Design Flow (CPLDs)

Design Entry and Synthesis

You can enter a design with a schematic editor or a text-based tool. Design entry begins with a design concept, expressed as a drawing or functional description. From the original design, a netlist is created, then synthesized and translated into a Native Generic Object (NGO) file. This file is fed into a program called NGDBuild, which produces a logical Native Generic Database (NGD) file.

The following figure shows the design entry and synthesis process.

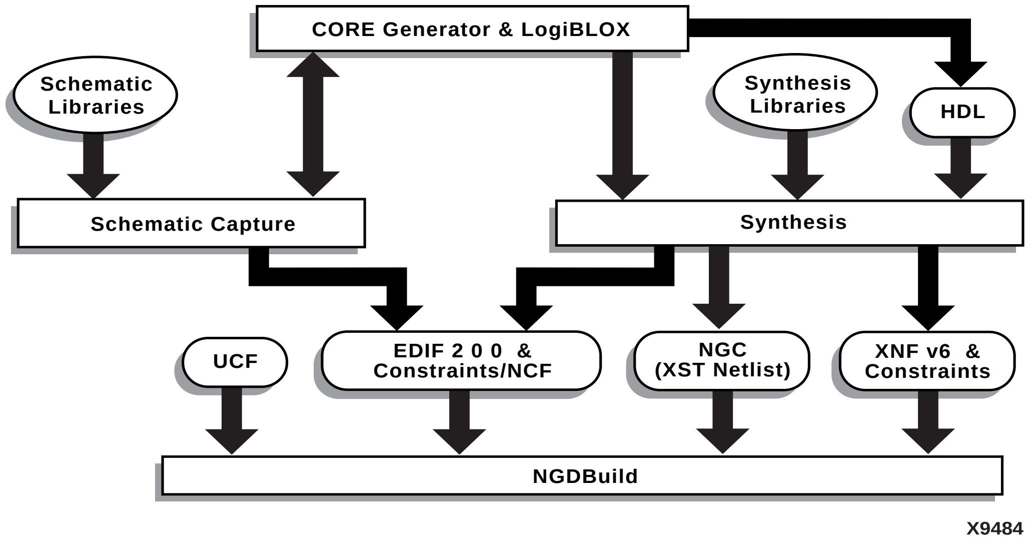


Figure 2-4 Design Entry Flow

Hierarchical Design

Design hierarchy is important in both schematic and HDL entry for the following reasons:

- Helps you conceptualize your design
- Adds structure to your design
- Promotes easier design debugging
- Makes it easier to combine different design entry methods (schematic, HDL, or state editor) for different parts of your design
- Makes it easier to design incrementally, which consists of designing, implementing, and verifying individual parts of a design in stages

- Reduces optimization time
- Facilitates concurrent design, which is the process of dividing a design among a number of people who develop different parts of the design in parallel, such as in Modular Design

A specific hierarchical name identifies each library element, unique block, and instance you create. The following example shows a hierarchical name with a 2-input OR gate in the first instance of a multiplexer in a 4-bit counter:

```
/Acc/alu_1/mult_4/8count_3/4bit_0/mux_1/or2
```

Xilinx strongly recommends that you name the components and nets in your design. These names are preserved and used by the FPGA Editor. These names are also used for back-annotation and appear in the debug and analysis tools. If you do not name your components and nets, the schematic editor automatically generates the names. For example, if left unnamed, the software might name the previous example as follows:

```
/$1a123/$1b942/$1c23/$1d235/$1e121/$1g123/$1h57
```

Note It is difficult to analyze circuits with automatically generated names, because they only have meaning for Xilinx software.

Schematic Entry Overview

Schematic tools provide a graphic interface for design entry. You can use these tools to connect symbols representing the logic components in your design. You can build your design with individual gates, or you can combine gates to create functional blocks. This section focuses on ways to enter functional blocks using library elements and the CORE Generator and LogiBLOX tools.

Library Elements

Primitives and macros are the “building blocks” of component libraries. Xilinx libraries provide primitives as well as common high-level macro functions. Primitives are basic circuit elements, such as AND and OR gates. Each primitive has a unique library name, symbol, and description. Macros contain multiple library elements, which can include primitives and other macros.

You can use the following types of macros with Xilinx FPGAs:

- Soft macros have pre-defined functionality, but have flexible mapping, placement, and routing. Soft macros are available for all FPGAs.
- Relationally Placed Macros (RPMs) have fixed mapping and relative placement. RPMs are available for all device families except the XC9500 family.

Macros are not available for synthesis because synthesis tools have their own module generators and do not require RPMs. If you wish to override the module generation, you can instantiate CORE Generator or LogiBLOX modules. For most leading-edge synthesis tools, this does not offer an advantage unless it is for a module that cannot be inferred.

CORE Generator Tool (FPGAs Only)

The Xilinx CORE Generator design tool delivers parameterizable cores that are optimized for Xilinx FPGAs. The library includes cores ranging from simple delay elements to complex DSP (Digital Signal Processing) filters and multiplexers. For details, refer to the *CORE Generator Guide*. You can also refer to the Xilinx IP (Intellectual Property) Center Web site at http://www.xilinx.com/ipcenter, which offers the latest IP solutions. These solutions include design reuse tools, free reference designs, DSP and PCI solutions, IP implementation tools, cores, specialized system level services, and vertical application IP solutions.

LogiBLOX Tool

The LogiBLOX tool generates a variety of variable-sized MSI- and LSI-level design building blocks, such as adders, counters, decoders, and shift registers. These modules complement the Xilinx macro libraries, which contain simpler, fixed-size logic and gate functions. The LogiBLOX tool also integrates these modules into your design. For further information, see the *LogiBLOX Guide*.

Note The LogiBLOX tool does not support the Virtex, Virtex-II, Virtex-E, and Spartan-II FPGA families. For these families, use the CORE Generator tool.

HDL Entry and Synthesis

A typical Hardware Description Language (HDL) supports a mixed-level description in which gate and netlist constructs are used with functional descriptions. This mixed-level capability enables you to describe system architectures at a high level of abstraction, then incrementally refine a design’s detailed gate-level implementation.

HDL descriptions offer the following advantages:

- You can verify design functionality early in the design process. A design written as an HDL description can be simulated immediately. Design simulation at this high level, at the gate-level before implementation, allows you to evaluate architectural and design decisions.
- An HDL description is more easily read and understood than a netlist or schematic description. HDL descriptions provide technology-independent documentation of a design and its functionality. Because the initial HDL design description is technology independent, you can use it again to generate the design in a different technology, without having to translate it from the original technology.
- Large designs are easier to handle with HDL tools than schematic tools.

After you create your HDL design, you must synthesize it. During synthesis, behavioral information in the HDL file is translated into a structural netlist, and the design is optimized for a Xilinx device. Xilinx supports HDL synthesis tools for several third-party synthesis vendor partners. In addition, Xilinx offers its own synthesis tool, Xilinx Synthesis Technology (XST). See the *Xilinx Synthesis Technology (XST) User Guide* for information. For detailed information on synthesis, see the *Synthesis and Simulation Design Guide*.

Functional Simulation

After you enter your design, you can simulate it. Functional simulation tests the logic in your design to determine if it works properly. You can save time during subsequent design steps if you perform functional simulation early in the design flow. See the “Simulation” section for more information.

Constraints

You may want to constrain your design within certain timing or placement parameters. You can specify mapping, block placement, and timing specifications.

You can enter constraints by hand or use the Constraints Editor, Floorplanner, or FPGA Editor. You can use the Timing Analyzer Graphical User Interface (GUI) or TRACE command line program to evaluate the circuit against these constraints. See the "TRACE" chapter and the online Help provided with each GUI for information. See the *Constraints Guide* for detailed information on constraints.

Mapping Constraints (FPGAs Only)

You can specify how a block of logic is mapped into CLBs using an FMAP or HMAP for all XC4000 and Spartan FPGA families or an FMAP for all Virtex FPGA families. These mapping symbols can be used in your schematic. However, if you overuse these specifications, it may be difficult to route your design.

Block Placement

Block placement can be constrained to a specific location, to one of multiple locations, or to a location range. Locations can be specified in the schematic, with synthesis tools, or in the User Constraint File (UCF). Poor block placement can adversely affect both the placement and the routing of a design. Typically, only I/O blocks require placement to meet external pin requirements.

Timing Specifications

You can specify timing requirements for paths in your design. PAR uses these timing specifications to achieve optimum performance when placing and routing your design.

Netlist Translation Programs

Two netlist translation programs allow you to read netlists into the Xilinx software tools. EDIF2NGD allows you to read an Electronic Data Interchange Format (EDIF) 2 0 0 file. XNF2NGD allows you to read an Xilinx Netlist Format (XNF). The NGDBuild program automatically invokes these programs as needed to convert your EDIF or XNF file to an NGD file, the required format for the Xilinx software tools. NGC files output from the Xilinx XST synthesis tool are read in by NGDBuild directly.

You can find detailed descriptions of the EDIF2NGD, XNF2NGD, and NGDBuild programs in the "NGDBuild" chapter and the "EDIF2NGD, XNF2NGD, and NGDBuild" appendix.

Design Implementation

Design Implementation begins with the mapping or fitting of a logical design file to a specific device and is complete when the physical design is successfully routed and a bitstream is generated. You can alter constraints during implementation just as you did during the Design Entry step. See the "Constraints" section for information.

The following figures show an overall view of the design implementation process for FPGAs and CPLDs.

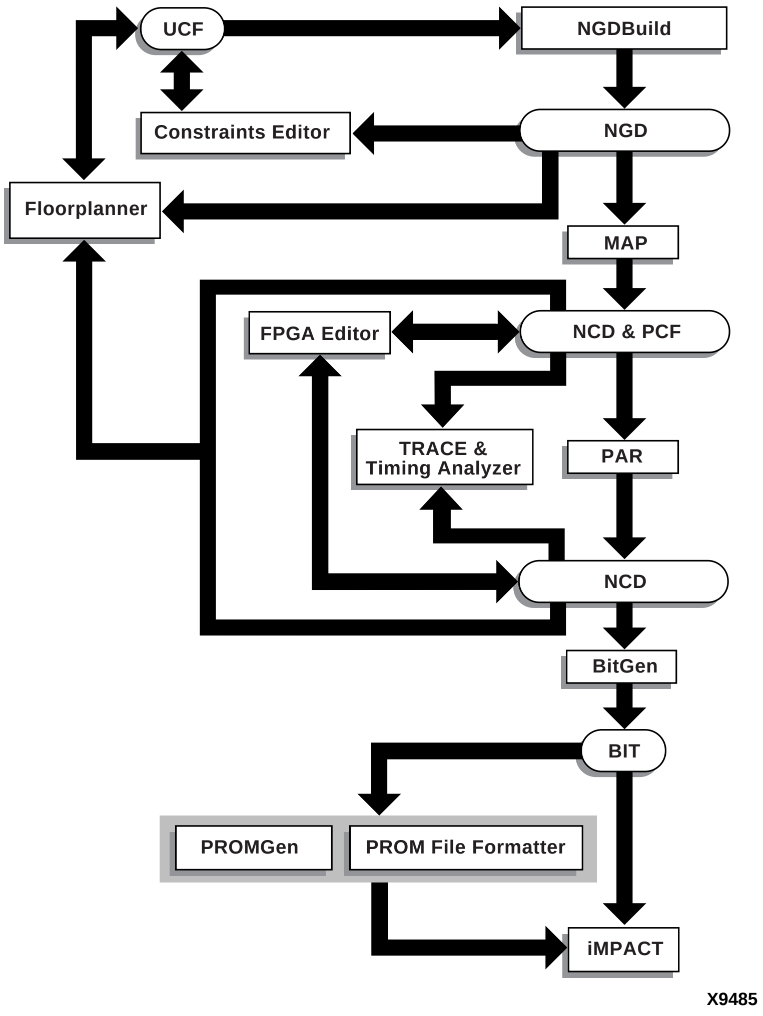


Figure 2-5 Design Implementation Flow (FPGAs)

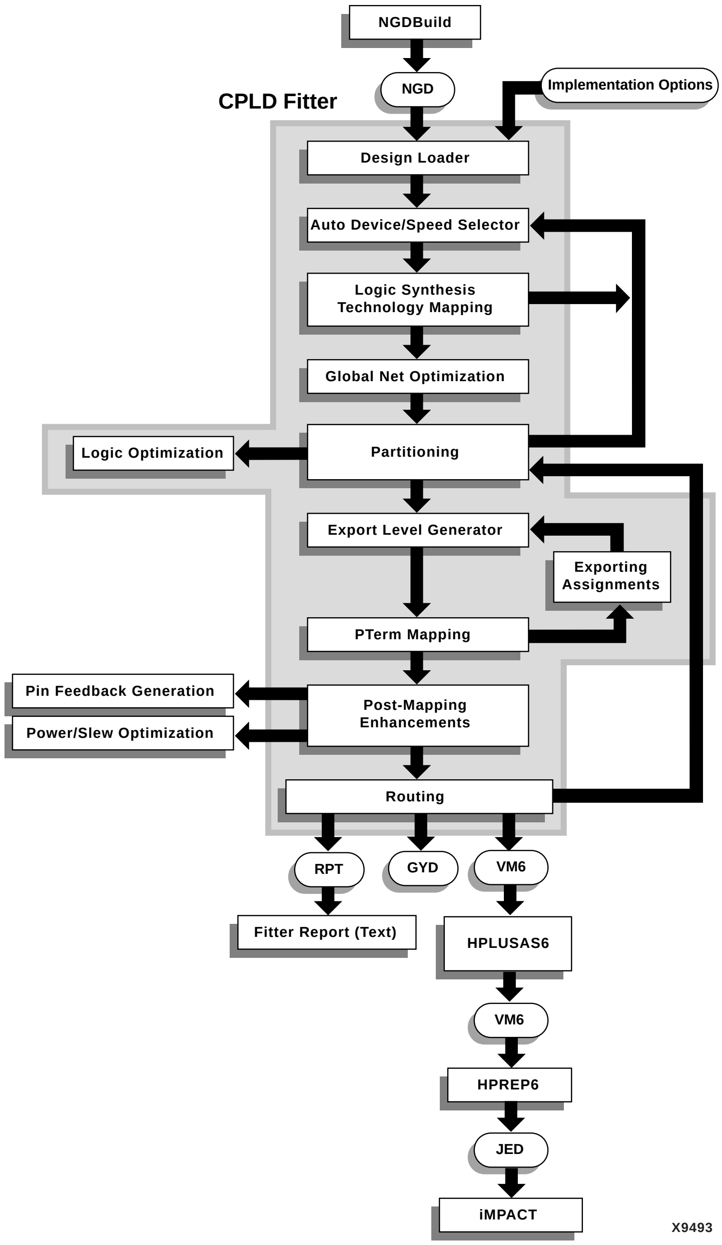


Figure 2-6 Design Implementation Flow (CPLDs)

Mapping (FPGAs Only)

For FPGAs, the MAP command line program maps a logical design to a Xilinx FPGA. The input to MAP is an NGD file, which contains a logical description of the design in terms of both the hierarchical components used to develop the design and the lower-level Xilinx primitives, and any number of NMC (hard placed-and-routed macro) files, each of which contains the definition of a physical macro. MAP then maps the logic to the components (logic cells, I/O cells, and other components) in the target Xilinx FPGA. The output design is an Native Circuit Description (NCD) file, which is a physical representation of the design mapped to the components in the Xilinx FPGA. The NCD file can then be placed and routed. See the "MAP" chapter for detailed information.

Placing and Routing (FPGAs Only)

For FPGAs, the PAR command line program takes an NCD file as input, places and routes the design, and outputs an NCD file, which is used by the bitstream generator, BitGen. The output NCD file can also act as a guide file when you reiterate placement and routing for a design to which minor changes have been made after the previous iteration. See the "PAR" chapter for detailed information.

You can also use the FPGA Editor GUI to do the following:

- Place and route critical components before running automatic place and route tools on an entire design
- Modify placement and routing manually; the editor allows both automatic and manual component placement and routing

Note For more information, see the online Help provided with the FPGA Editor.

Bitstream Generation (FPGAs Only)

For FPGAs, the BitGen command line program produces a bitstream for Xilinx device configuration. BitGen takes a fully routed NCD file as its input and produces a configuration bitstream—a binary file with a .bit extension. The BIT file contains all of the configuration information from the NCD file defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. See the “BitGen” chapter for detailed information.

After you generate your BIT file, you can download it to a device using the iMPACT GUI. You can also format the BIT file into a PROM file using the PromGen command line program or the Prom File Formatter GUI, and then download it to a device using the iMPACT GUI. See the “PROMGen” chapter of this guide, the *PROM File Formatter Guide*, or the *iMPACT User Guide* for more information.

Design Verification

Design verification is the process of testing the functionality and performance of your design. You can verify Xilinx designs in the following ways:

- Simulation (functional and timing)
- Static timing analysis
- In-circuit verification

The following tables lists the different design tools used for each verification type.

Table 2-1 Verification Tools

Verification Type	Tools
Simulation	Third party simulators (integrated and non-integrated)
Static timing analysis	TRACE (command line program) Timing Analyzer (GUI) Mentor Graphics® TAU and Innoveda BLAST software for use with the STAMP file format (for I/O timing verification only)
In-Circuit Verification	Design Rule Checker (command line program) Download cable

Design verification procedures should occur throughout your design process, as shown in the following figures.

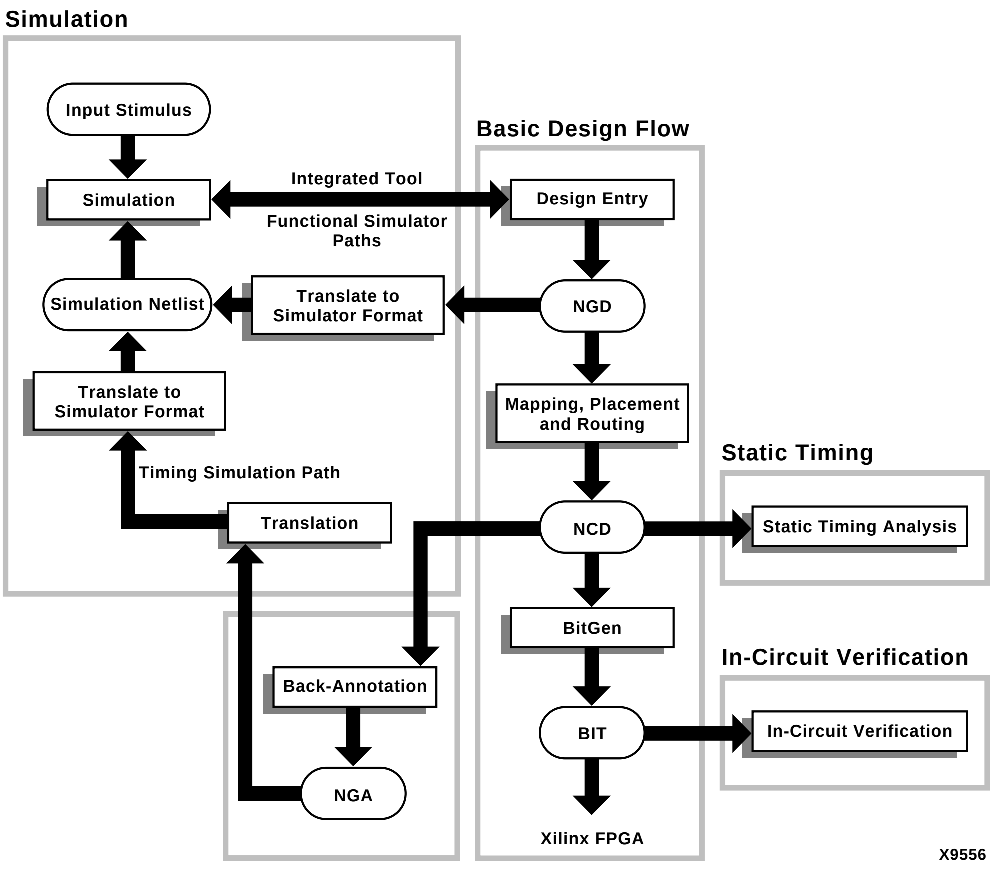


Figure 2-7 Three Verification Methods of the Design Flow (FPGAs)

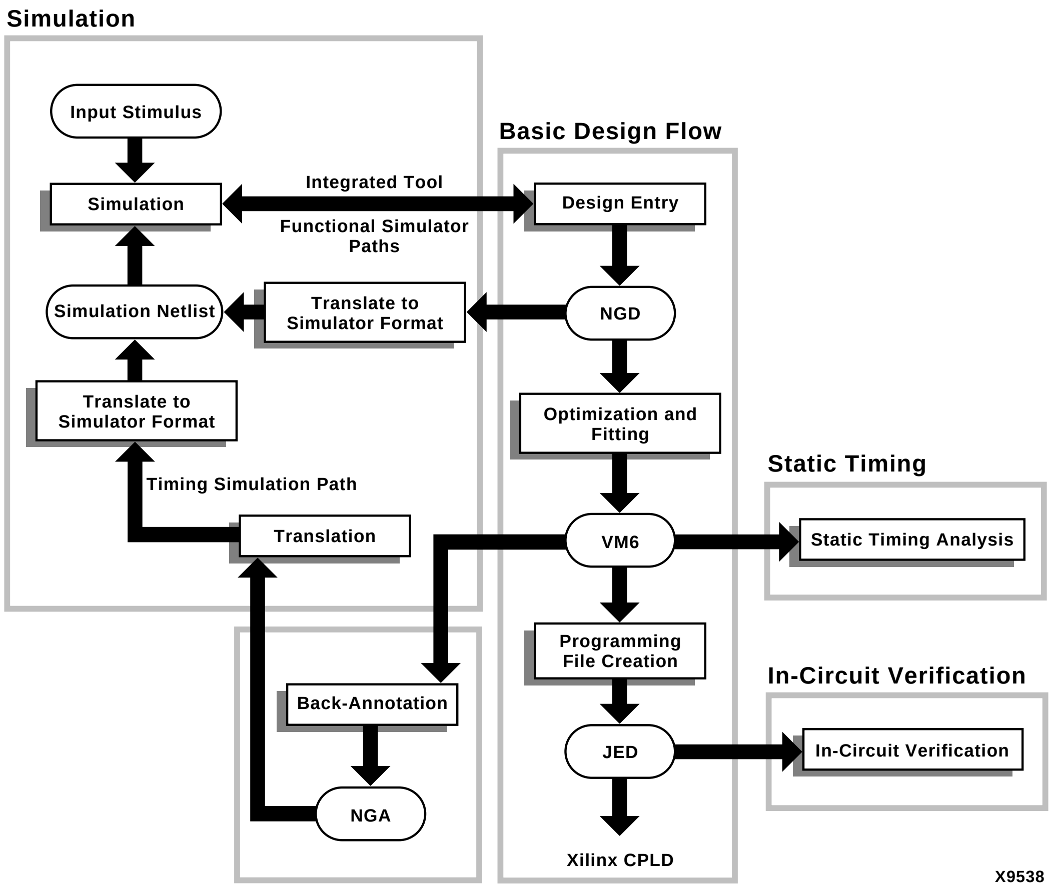


Figure 2-8 Three Verification Methods of the Design Flow (CPLDs)

Simulation

You can run functional or timing simulation to verify your design. This section describes the back-annotation process that must occur prior to timing simulation. It also describes the functional and timing simulation methods for both schematic and HDL-based designs.

Back-Annotation

Before timing simulation can occur, the physical design information must be translated and distributed back to the logical design. For FPGAs, this back-annotation process is done with a program called NGDAnno. For CPLDs, back-annotation is performed with the TSim Timing Simulator. These programs create a database for the netlist writers, which translate the back-annotated information into a netlist format that can be used for timing simulation. The following figures show the back-annotation flows.

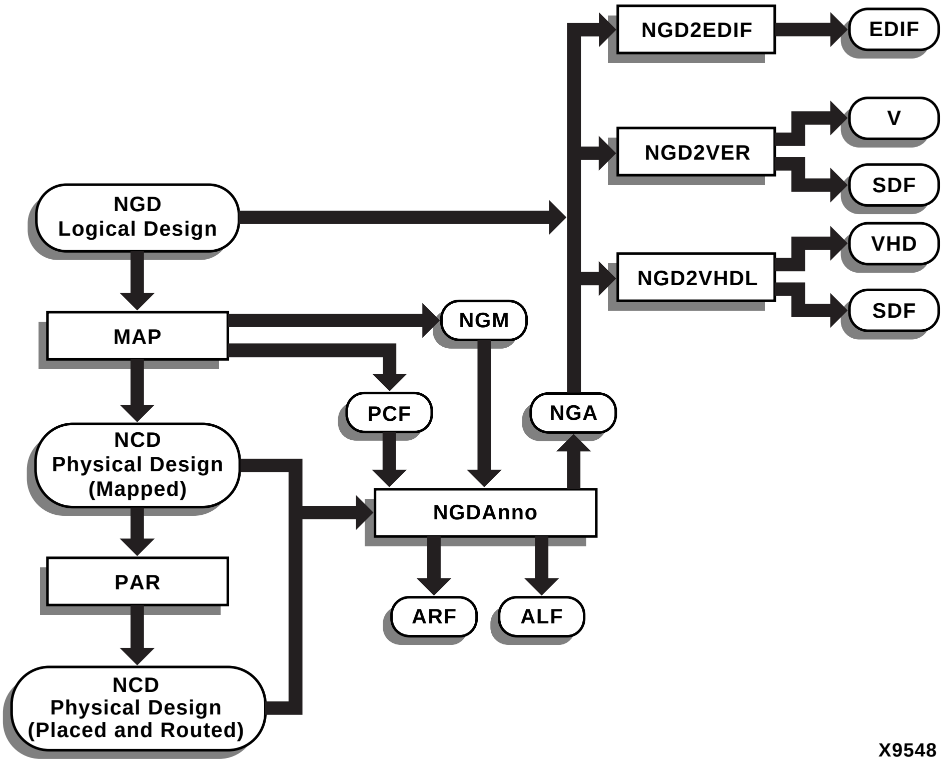


Figure 2-9 Back-Annotation (FPGAs)

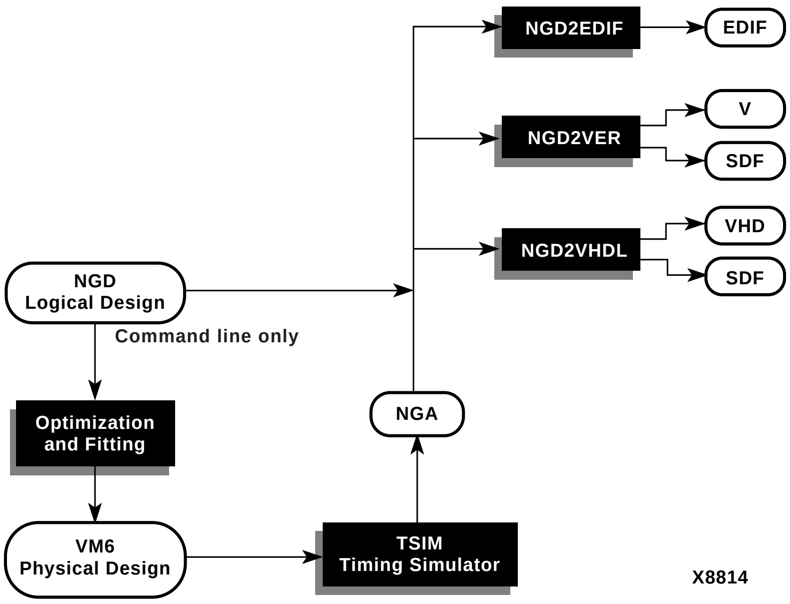


Figure 2-10 Back-Annotation (CPLDs)

NGDAnno (FPGAs Only)

NGDAnno is a command line program that distributes information about delays, setup and hold times, clock to out, and pulse widths found in the physical NCD design file back to the logical NGD file. NGDAnno reads an NCD file as input. The NCD file can be a mapped-only design, or a partial or fully placed and routed design.

An NGM file, created by MAP, is an optional source of input. NGDAnno merges mapping information from the NGM file with placement, routing, and timing information from the NCD file.

NGDAnno outputs a Native Generic Annotated (NGA) file, which is a back-annotated NGD file. This file is input to the appropriate netlist writer, which converts the binary Xilinx database format back to an ASCII netlist. See the "NGDAnno" chapter for detailed information.

Note Use caution when making changes to the functional behavior of your design. For example, if you make logical changes to an NCD design from within the FPGA Editor, NGDAnno is unable to correlate the changed objects in the physical design with the objects in the logical design. Instead, it recreates the entire NGA design from the NCD and issues a warning indicating that the NCD does not match the NGM.

Netlist Writers

Netlist writers (NGD2EDIF, NGD2VER, or NGD2VHDL) take the output of NGDAnno or the CPLD command and create a simulation netlist in the specified format. An NGD or NGA file is input to each of the netlist writers. The NGD file is a logical design file containing primitive components, while the NGA file is a back-annotated logical design file. Following is a list of the supported netlist writers with descriptions of their input and output files:

- NGD2EDIF translates an NGD or NGA file into an EDIF netlist. Output from the NGD2EDIF program is an EDN file, a netlist in EDIF format. The default EDN file produced by NGD2EDIF is generic. If you want to produce EDIF targeted to Mentor Graphics or Innoveda, you must include the –v (vendor) option. See the "NGD2EDIF" chapter for detailed information.
- NGD2VER translates an NGD or NGA file into a Verilog netlist (V) file. If the input is an NGA file, NGD2VER also generates an SDF (Standard Delay Format) file. The resulting V and SDF files have the same root name as the NGD or NGA file unless you specify otherwise. See the "NGD2VER" chapter for detailed information.

  Note The SDF file contains timing information intended solely for use with the Verilog file which was generated from the same NGA file. Do not attempt to use the SDF file in conjunction with the original Verilog netlist design or the product of another netlist writer.
- NGD2VHDL translates an NGD or NGA file into a VHDL netlist (VHD). If the input file is an NGA file, NGD2VHDL also generate an SDF (Standard Delay Format) file. The VHD and SDF files have a time_sim root name by default. See the "NGD2VHDL" chapter for detailed information.

Note The netlist writers also allow you to verify that the circuit logic is correct *before* you implement the design. You can use the data in a non-implemented NGD design as input to a netlist writer. You can then run a simulation program on the resulting netlist.

Schematic-Based Simulation

Design simulation involves testing your design using software models. It is most effective when testing the functionality of your design and its performance under worst-case conditions. You can easily probe internal nodes to check your circuit's behavior, and then use these results to make changes in your schematic.

Simulation is performed using third-party tools that are linked to the Xilinx Development System. Use the various CAE-specific interface user guides, which cover the commands and features of the Xilinx-supported simulators, as your primary reference.

The software models provided for your simulation tools are designed to perform detailed characterization of your design. You can perform functional or timing simulation, as described in the following sections.

Functional Simulation

Functional simulation determines if the logic in your design is correct before you implement it in a device. Functional simulation can take place at the earliest stages of the design flow. Because timing information for the implemented design is not available at this stage, the simulator tests the logic in the design using unit delays.

Note It is usually faster and easier to correct design errors if you perform functional simulation early in the design flow.

You can use integrated and non-integrated simulation tools. Integrated tools, such as Mentor or Innoveda, often contain a built-in interface that links the simulator and a schematic editor, allowing the tools to use the same netlist. You can move directly from entry to simulation when using a set of integrated tools.

Functional simulation in schematic-based tools is performed immediately after design entry in the capture environment. The schematic capture tool requires a Xilinx Unified Library and the simulator requires a library if the tools are not integrated. Most of the schematic-based tools require translation from their native database to XNF or EDIF for implementation. The return path from implementation is usually EDIF with certain exceptions in which a schematic tool is tied to an HDL simulator.

Timing Simulation

Timing simulation verifies that your design runs at the desired speed for your device under worst-case conditions. This process is performed after your design is mapped, placed, and routed for FPGAs or fitted for CPLDs. At this time, all design delays are known.

Timing simulation is valuable because it can verify timing relationships and determine the critical paths for the design under worst-case conditions. It can also determine whether or not the design contains set-up or hold violations.

Before you can simulate your design, you must go through the back-annotation process, as described in the "Back-Annotation" section. During this process, the Xilinx netlist writers create suitable formats for various simulators.

Note Naming the nets during your design entry is important for both functional and timing simulation. This allows you to find the nets in the simulations more easily than looking for a software-generated name.

HDL-Based Simulation

Xilinx supports functional and timing simulation of HDL designs at the following points:

- Register Transfer Level (RTL) simulation which may include the following:
    - Instantiated UniSim library components
    - LogiBLOX modules
    - LogiCORE models
- Post-synthesis functional simulation with one of the following:
    - Gate-level UniSim library components
    - Gate-level pre-route SimPrim library components
- Post-implementation back-annotated timing simulation with the following:
    - SimPrim library components
    - Standard Delay Format (SDF) file

The following figure shows when you can perform functional and timing simulation.

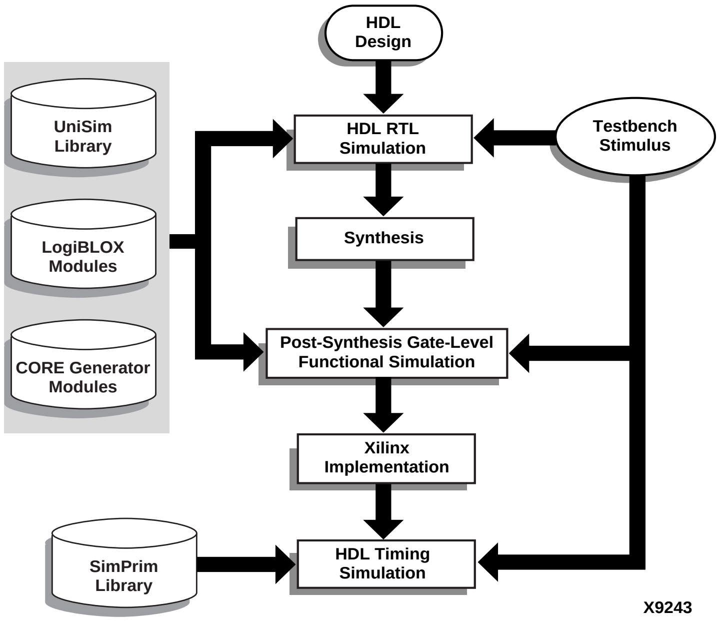


Figure 2-11 Simulation Points for HDL Designs

The three primary simulation points can be expanded to allow for two post-synthesis simulations. These points can be used if the synthesis tool cannot write VHDL or Verilog, or if the netlist is not in terms of UniSim components. The following table lists all the simulation points available in the HDL design flow.

Table 2-2 Five Simulation Points in HDL Design Flow

Simulation	UniSim	LogiBLOX Models	SimPrim	SDF
RTL	X	X		
Post-Synthesis	X	X		
Functional Post-NGDBuild (Optional)			X	
Functional Post-MAP (Optional)			X	X
Post-Route Timing			X	X

These simulation points are described in the "Simulation Points" section of the *Synthesis and Simulation Design Guide*.

The libraries required to support the simulation flows are described in detail in the "VHDL / Verilog Libraries and Models" section of the *Synthesis and Simulation Design Guide*. The flows and libraries support close functional equivalence of initialization behavior between functional and timing simulations. This is due to the addition of new methodologies and library cells to simulate Global Set / Reset (GSR) and Global 3-State (GTS) behavior.

You must address the built-in reset circuitry behavior in your designs, starting with the first simulation, to ensure that the simulations agree at the three primary points. If you do not simulate GSR behavior prior to synthesis and place and route, your RTL and post-synthesis simulations may not initialize to the same state as your post-route timing simulation. If this occurs, your various design descriptions are not functionally equivalent and your simulation results do not match.

In addition to the behavioral representation for GSR, you must add a Xilinx implementation directive. This directive is used to specify to the place and route tools to use the special purpose GSR net that is pre-routed on the chip, and not to use the local asynchronous set / reset pins. Some synthesis tools can identify the GSR net from the behavioral description, and place the STARTUP module on the net to direct the implementation tools to use the global network. However, other synthesis tools interpret behavioral descriptions literally and introduce additional logic into your design to implement a function. Without specific instructions to use device global networks, the Xilinx implementation tools use general-purpose logic and interconnect resources to redundantly build functions already provided by the silicon.

Even if GSR behavior is not described, the chip initializes during configuration, and the post-route netlist has a net that must be driven during simulation. The "Understanding the Global Signals for Simulation" section of the *Synthesis and Simulation Design Guide* includes the methodology to describe this behavior, as well as the GTS behavior for output buffers.

Xilinx VHDL simulation supports the VITAL standard. This standard allows you to simulate with any VITAL-compliant simulator. Built-in Verilog support allows you to simulate with the Cadence Verilog-XL and other compatible simulators. Xilinx HDL simulation supports all current Xilinx FPGA and CPLD devices. Refer to the *Synthesis and Simulation Design Guide* for the list of supported VHDL and Verilog standards.

Static Timing Analysis (FPGAs Only)

Static timing analysis is best for quick timing checks of a design after it is placed and routed. It also allows you to determine path delays in your design. Following are the two major goals of static timing analysis:

- Timing verification

  This is the process of verifying that the design meets your timing constraints.

- Reporting

  This is the process of enumerating input constraint violations and placing them into an accessible file. You can analyze partially or completely placed and routed designs. The timing information depends on the placement and routing of the input design.

You can run static timing analysis using the Timing Reporter and Circuit Evaluator (TRACE) command line program. See the

"TRACE" chapter for detailed information. You can also use the Timing Analyzer GUI to perform this function. See the online Help provided with the Timing Analyzer. Use either tool to evaluate how well the place and route tools met the input timing constraints.

In-Circuit Verification

As a final test, you can verify how your design performs in the target application. In-circuit verification tests the circuit under typical operating conditions. Because you can program your Xilinx devices repeatedly, you can easily load different iterations of your design into your device and test it in-circuit. To verify your design in-circuit, download your design bitstream into a device with the Parallel Cable III or MultiLINX cable.

Note For information about Xilinx cables and hardware, see the *iMPACT User Guide.*

Design Rule Checker (FPGAs Only)

Before generating the final bitstream, it is important to use the DRC option in BitGen to evaluate the NCD file for problems that could prevent the design from functioning properly. DRC is invoked automatically unless you use the –d option. See the "BitGen" chapter and "Physical Design Rule Check" chapter for detailed information.

Xilinx Design Download Cables

Xilinx provides the Parallel Cable III or MultiLINX cable, depending on which development system you are using. To download your design, you must create a configuration bitstream.

For FPGAs, you can use the MultiLINX cable to read back and verify configuration data. Detailed cable connection and daisy-chain information is provided in the *iMPACT User Guide*.

Note You can use the Xilinx Parallel Cable III for FPGA and CPLD design download and readback, but the cable does not have a design verification function.

FPGA Design Tips

The Xilinx FPGA architecture is best suited for synchronous design. Strict synchronous design ensures that all registers are driven from the same time base with no clock skew. This section describes several tips for producing high-performance synchronous designs.

Design Size and Performance

Information about design size and performance can help you to optimize your design. When you place and route your design, the resulting report files list the number of CLBs, IOBs, and other device resources available. A first pass estimate can be obtained by processing the design through the MAP program.

If you want to determine the design size and performance without running automatic implementation software, you can quickly obtain an estimate from a rough calculation based on the Xilinx FPGA architecture. See *The Programmable Logic Data Book* for more information on all Xilinx FPGA architectures.

Global Clock Distribution

Xilinx clock networks guarantee small clock skew values. The following table lists the resources available for the Xilinx FPGA families.

Table 2-3 Global Clock Resources

FPGA Family	Resource	Number	Destination Pins
XC4000E/L	BUFGP BUFGS	4 4	Clock, control, or certain input Clock, control, or certain input
XC4000EX/XL/XLA	BUFG BUFGLS BUFGE BUFFCLK	8 8 8 4	Clock, control, or certain input Clock, control, or certain input Clock, control, or certain input I/O clock
Spartan	BUFGP BUFGS	4 4	Clock, control, or certain input Clock, control, or certain input
SpartanXL	BUFGLS	8	Clock, control, or certain input
Virtex, Virtex-E, Spartan-II	BUFG	4	Clock
Virtex-II	BUFGMUX	16	Clock

Note In certain devices families, global clock buffers are connected to control pin and logic inputs. If a design requires extensive routing, there may be extra routing delay to these loads.

Data Feedback and Clock Enable

The following figure shows a gated clock. The gated clock's corresponding timing diagram shows that this implementation can lead to clock glitches, which can cause the flip-flop to clock at the wrong time.

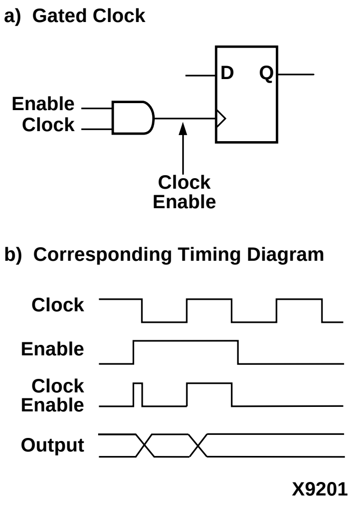


Figure 2-12 Gated Clock

The following figure shows a synchronous alternative to the gated clock using a data path. The flip-flop is clocked at every clock cycle and the data path is controlled by an enable. When the enable is Low, the multiplexer feeds the output of the register back on itself. When the enable is High, new data is fed to the flip-flop and the register changes its state. This circuit guarantees a minimum clock pulse width and it does not add skew to the clock. The XC4000, Spartan-II, and Virtex families' flip-flops have a built-in clock-enable (CE).

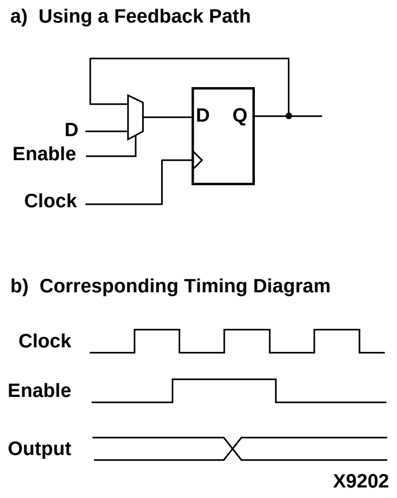


Figure 2-13 Synchronous Design Using Data Feedback

Counters

Cascading several small counters to create a larger counter is similar to a gated clock. For example, if two 8-bit counters are connected, the terminal counter (TC) of the first counter is a large AND function gating the second clock input. The following figure shows how you can create a synchronous design using the CE input. In this case, the TC of the first stage is connected directly to the CE of the second stage.

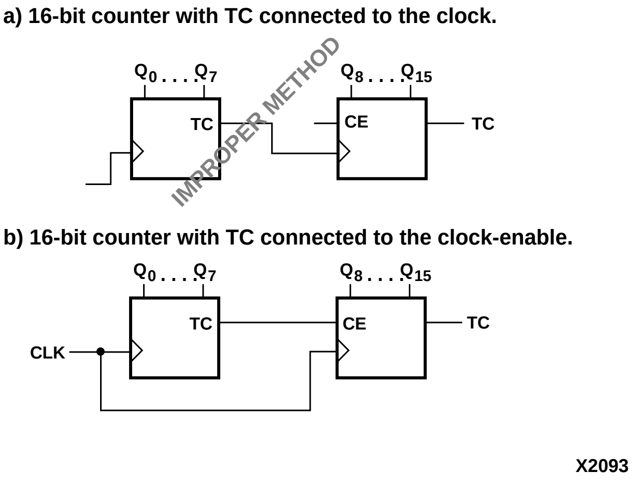

a) 16-bit counter with TC connected to the clock.
IMPROPER METHOD
Q0 . . . Q7
Q8 . . . Q15
TC
CE
TC
b) 16-bit counter with TC connected to the clock-enable.
Q0 . . . Q7
Q8 . . . Q15
TC
CE
TC
CLK
X2093

Figure 2-14 Two 8-Bit Counters Connected to Create a 16-Bit Counter

Other Synchronous Design Considerations

Other considerations for achieving a synchronous design include the following:

- Use clock enables instead of gated clocks to control the latching of data into registers.
- If your design has more clocks than the number of global clock distribution networks, try to redesign to reduce the number of clocks. Otherwise, put the clocks that have the lowest fanout onto normally routed nets, and specify a low MAXSKEW rating. A clock net routed through a normal net has skew.
- Use the Virtex low skew resources. Make sure the MAXSKEW rating is *not* specified when using these resources.

Chapter 3

Modular Design

Modular Design is compatible with the following device families:

- Virtex / -II / -II PRO / -E
- Spartan-II / -IIE

This chapter includes an overview of Modular Design and describes how to run the Modular Design flow. It contains the following sections:

Modular Design Overview

Modular Design allows a team of engineers to independently work on different pieces, or "modules," of a design and later merge these modules into one FPGA design. This parallel development saves time and allows for independent timing closure on each module. Modular Design also allows you to modify a module while leaving other, more stable modules intact.

Note Modular Design is a Xilinx Development System Option. Go to http://www.xilinx.com for information on obtaining this optional feature.

The Modular Design flow consists of the following steps:

- Modular Design Entry and Synthesis—In this step, the team creates designs using a Hardware Description Language (HDL) and synthesizes them. This step must be done for both the top-level design and the modules as follows:
    - Top-Level Design

      The team leader must complete design entry and synthesis for the top-level design before the Initial Budgeting phase of Modular Design Implementation can begin.
    - Modules

      Each team member must complete design entry and synthesis for his or her module before moving on to the Active Module Implementation phase for that module. Team members can complete module design entry and synthesis in parallel and can even complete this step while the team leader is working on the Initial Budgeting phase of Modular Design Implementation.
- Modular Design Implementation—This step comprises the following phases:
    - Initial Budgeting

      In this phase, the team leader assigns top-level constraints to the top-level design.

      Note Initial Budgeting is not required for modules.
    - Active Module Implementation

      In this phase, the team members implement the modules.
    - Final Assembly

      In this phase, the team leader assembles the top-level design and the implemented modules into one design.

The following figure shows the Modular Design flow.

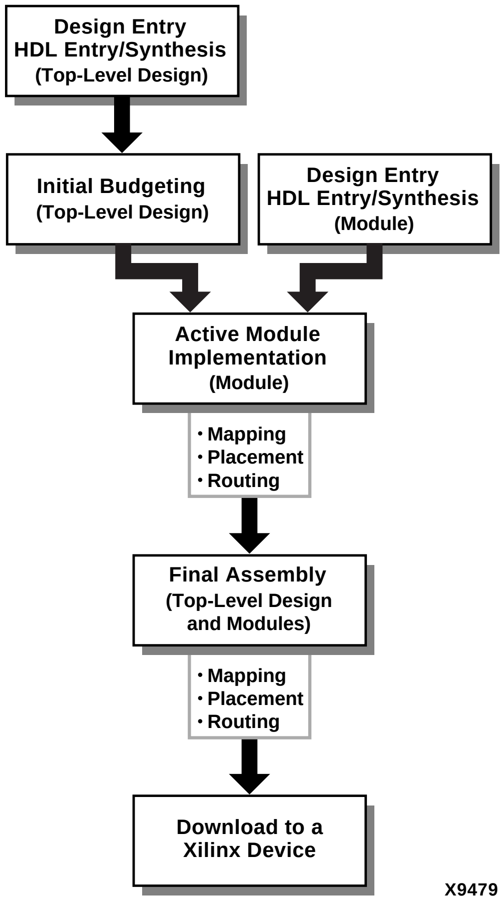


Figure 3-1 Modular Design Flow Overview

Modular Design requires up front planning to ensure that the design is partitioned properly. It also requires communication among team members to ensure that partitions work together during the Final Assembly phase. The number of modules should not be greater than the number of team members. Modular Design is best used for large designs that can easily be partitioned into self-contained modules.

Note Modular Design is *not* recommended for the direct conversion of large ASIC designs that contain heavily interconnected logic. Such designs cannot be easily partitioned into independent modules.

Modular Design Entry and Synthesis

The team leader creates the top-level design using an HDL and synthesizes this design. The top-level design includes all global logic, all modules instantiated as “black boxes” with only ports and port directions, and the signals that connect modules to each other and to I/O ports. This step must occur before Modular Design Implementation can begin.

The team members create individual module designs using an HDL and synthesize the designs. However, this does not need to occur before the Modular Design Implementation step begins. Team members can work on their module designs while the team leader moves on to the Initial Budgeting phase of Modular Design Implementation. Team members must complete design entry and synthesis for their modules prior to the Active Module Implementation phase of Modular Design Implementation.

You can enter your design with a text-based tool using Verilog or VHDL. To synthesize your design, you can use Xilinx-supported third-party tools, which produce a design file in third party netlist formats, or you can use the Xilinx synthesis tool, Xilinx Synthesis Technology (XST). For more information on XST, see the *Xilinx Synthesis Technology (XST) User Guide*.

As with standard design entry, Modular Design entry begins with a design concept, expressed as a functional description. From the original design, a netlist is created. For details on HDL design entry, see the “HDL Entry and Synthesis” section of the “Design Flow” chapter. Also, see the *Synthesis and Simulation Design Guide*.

The following figure shows the Modular Design entry and synthesis flow.

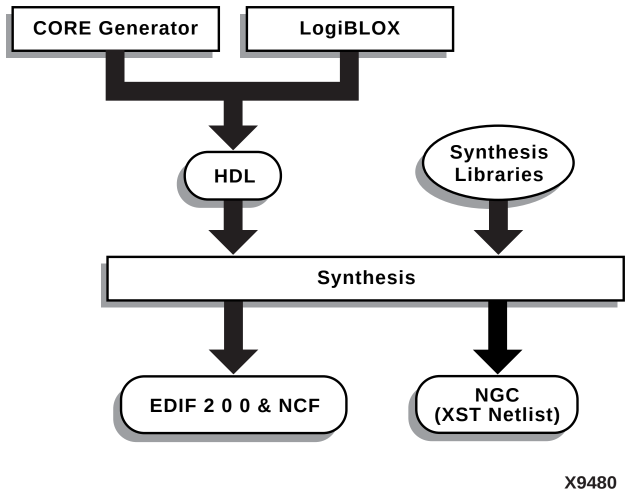


Figure 3-2 Modular Design Entry and Synthesis Flow

Modular Design Implementation

Modular Design implementation includes the three phases described in this section. After the final phase is complete, you can use the implemented design to generate a bitstream.

Initial Budgeting Phase

In this phase, the team leader positions the top-level logic for the design. Properly positioning the logic in this phase is critical. Repositioning top-level logic later in the design process requires that you rerun each phase of the Modular Design flow, which is time consuming. Following are the objectives of the Initial Budgeting phase:

- Position global logic
- Size and position each module on the target chip
- Position the input and output ports for each module
- Budget initial timing constraints

The first step in this phase is to run NGDBuild in initial budgeting mode. NGDBuild generates an NGD file with all of the instantiated modules represented as unexpanded blocks. This NGD file cannot be mapped but can be used with the Constraints Editor and Floorplanner. Using the Constraints Editor, you assign timing constraints to the top-level design. Using the Floorplanner, you assign location constraints for each module.

You must position *all* top-level logic during this phase. This ensures that the remainder of the logic for the design is optimally positioned during the Active Module Implementation phase. Location constraints must be assigned for the following elements:

- Top-level I/O ports

  These are the top-level ports of the design that are mapped into IOB components. In a typical design, these IOB locations are already well defined because of board layout requirements.

- Top-level logic

  This is logic positioned at the top-level of a design, such as global buffers, 3-state buffers, flip-flops, and look-up tables. In a typical design, there is only a small amount of top-level logic.

  When positioning BUFTs, follow these rules:

  - If more than one BUFT is driving the same net, position the BUFTs in the same row
  - If BUFTs are driving different nets, position each in a different row
  - If there are multiple 3-state nets and each 3-state net is driven by multiple BUFTs, position one BUFT for each 3-state net

- Each module

  Estimate how many resources each module will take and position each module accordingly. Xilinx recommends that you allow space between modules so you can position the module ports.

- “Pseudo logic” that represents module ports

  When two modules are connected at this stage in the design flow, they are not connected directly. Instead, each of the modules has a module port that is connected to pseudo logic. This pseudo logic is either a “pseudo load” or a “pseudo driver.” A pseudo load is a temporary load for the module's output, because its actual load is located in an unexpanded module. A pseudo driver is a temporary driver for a module, because its actual driver is located in an unexpanded module.

The following figure shows the relationship between the modules and pseudo logic during the Initial Budgeting phase.

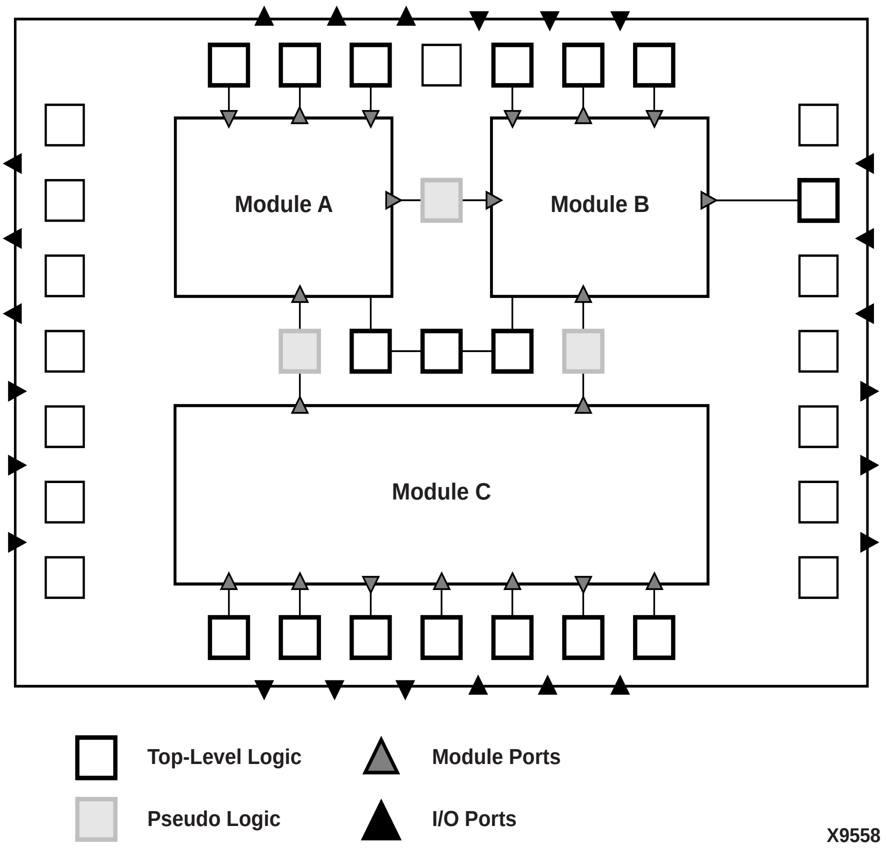


Figure 3-3 Modules and Pseudo Logic

The following figure shows a detailed view of the pseudo logic shown in the preceding figure. In the following figure, the output port for unexpanded Module A is connected to a pseudo load, and the input port for Module B is connected to a pseudo driver.

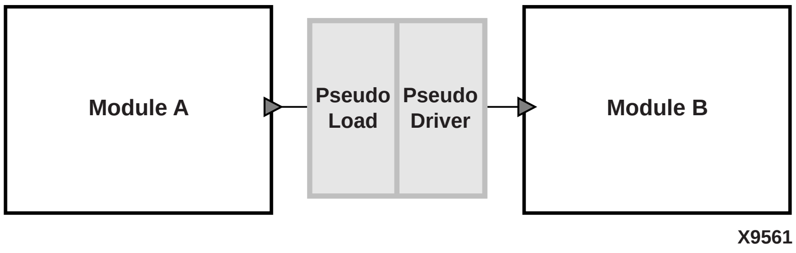


Figure 3-4 Pseudo Driver and Pseudo Load

The term "pseudo logic" is used, because it is logic that is temporarily inserted to facilitate the relative placement of the connected logic within a module. In the final assembled design, the pseudo logic does not appear, instead the actual logic is directly connected.

Note Pseudo logic is only created on a net that connects two modules. If a net connects a module to top-level logic, pseudo logic is unnecessary, because the top-level logic constrains the module logic.

The following figure shows the flow through the Initial Budgeting phase.

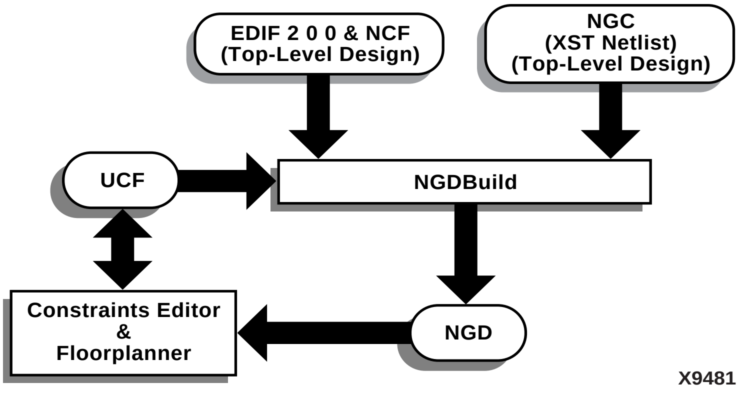


Figure 3-5 Initial Budgeting Flow

Active Module Implementation Phase

In this phase, team members implement the top-level design with one module expanded at a time. This must be done separately for each module and takes place in the individual module directories, rather than at the top-level directory. Active Module Implementation can be done in parallel, that is each team member can implement his or her module at the same time.

To accomplish this, you run NGDBuild in active module mode. NGDBuild reads the top-level design, the module netlist, and the top-level UCF file as input. NGDBuild generates the top-level design as an NGD file (*design_name*.ngd) with just the specified "active" module expanded. At this point, you can use the Constraints Editor to apply timing constraints for the active module. You can then map, place, and route the NGD file.

The placed and routed designs are called Physically Implemented Modules (PIMs). The PIMCreate utility automatically copies the PIM's NGO, NCD, and NGM files to the appropriate directory in preparation for the Final Assembly phase. It also renames the files *design_name*.ncd and *design_name*.ngm to *module_name*.ncd and *module_name*.ngm as needed for later phases of Modular Design.

The following figure shows the flow through the Active Module Implementation phase.

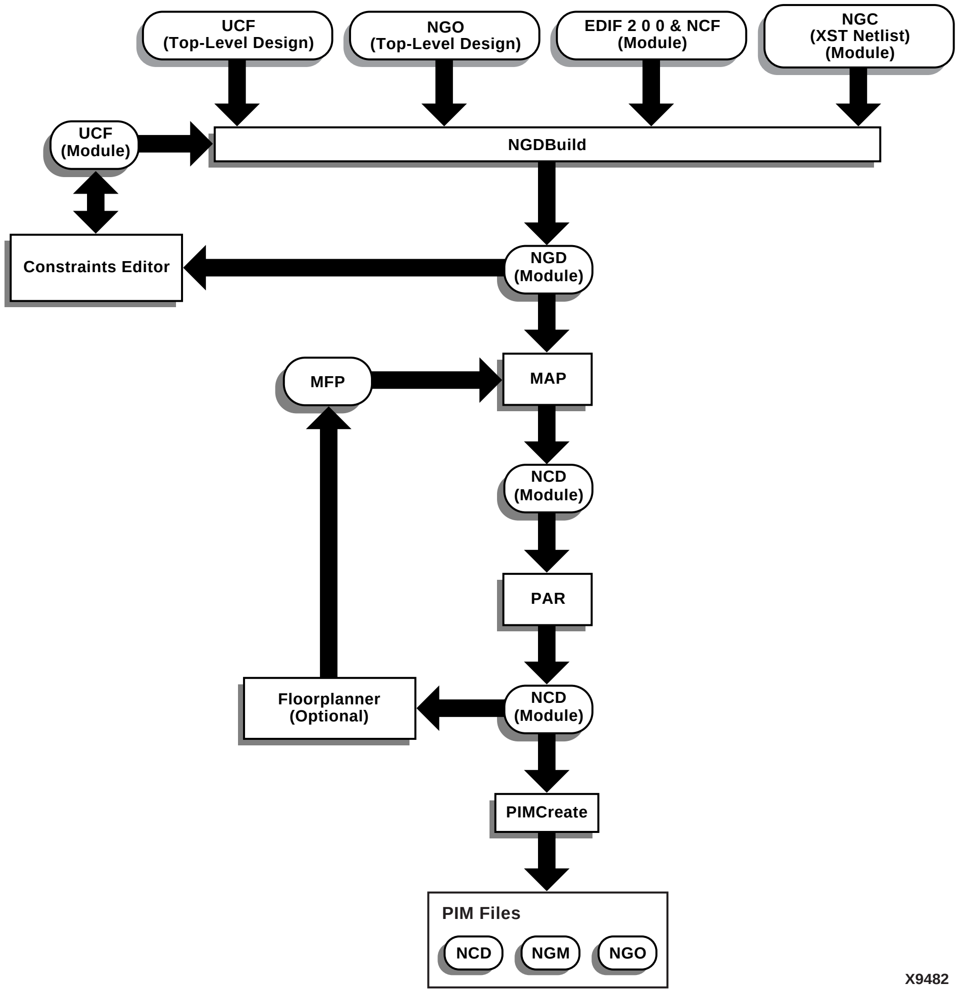


Figure 3-6 Active Module Implementation Flow

Final Assembly Phase

In this phase, the team leader assembles the modules into one design by running NGDBuild in final assembly mode. NGDBuild reads in the top-level NGO file, the top-level UCF file, and all of the PIMs to create a fully expanded design file that you can map, place, and route. During this phase, the place and route tools copy the placement and routing information from each PIM. This preserves the exact timing performance from the Active Module Implementation phase for each module in the design.

The following figure shows the flow through the Final Assembly phase.

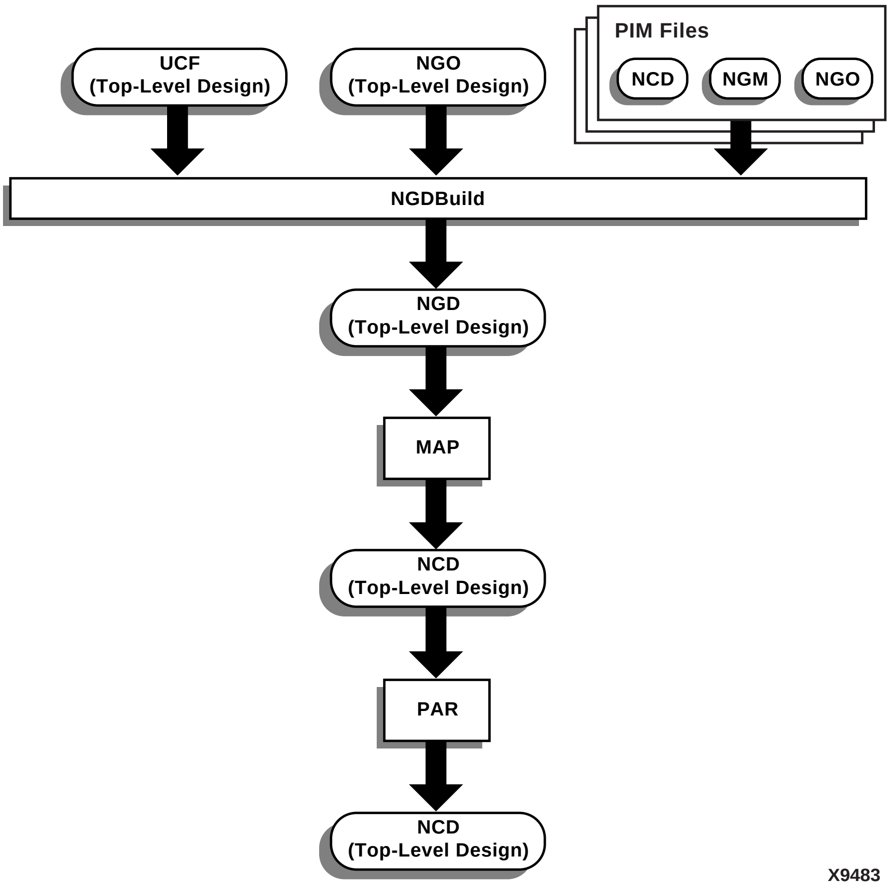


Figure 3-7 Final Assembly Flow

Setting Up Modular Design Directories

Before the team starts designing, it is essential that the team leader sets up an organized directory structure that works for the team. These directories are used for synthesis of the top-level design and during the Initial Budgeting and Final Assembly phases. Following is the recommended directory structure for the standard Modular Design flow:

- “Synthesis” directory

  This directory should contain a directory for the top-level design. The team leader synthesizes the top-level design in the top-level design directory. The top-level design directory should include the appropriate HDL file, the project file, and project directories.

  Note Synthesis of individual modules can take place in the team members’ local directories. Xilinx recommends setting up a directory to synthesize your module that is separate from the directory used to implement your module. The synthesis directory should include the appropriate HDL file, the project file, and project directories.

- “Implementation” directory

  This directory should contain the following:

  - Directory for the top-level design

    The team leader sets up initial budgeting for the design in this directory. After team members publish the implemented modules to the PIMs directory, the team leader also assembles the top-level design and PIMs into the final design in this directory.

  - Directory for the PIMs

    The PIMs directory stores the implemented module files. When a team member runs the PIMCreate utility during the Active Module Implementation phase, PIMCreate creates the appropriate module directory in the PIMs directory and copies the implemented module files to the module directory.

  Note Implementation of individual modules can take place in the team members’ local directories. However, each implemented module must be published to the PIMs directory using the PIMCreate command line tool.

Running the Standard Modular Design Flow

Use the standard Modular Design flow for most designs. This flow requires that *all* modules are implemented and published to the PIMs directory before they are assembled together. After you are comfortable running this flow, you can also run it "sequentially." For more information, see the "Running the Sequential Modular Design Flow" section.

Entering the Design

Create your top-level and module designs with a text-based tool using Verilog or VHDL. Use the following guidelines when creating your code.

General Coding Guidelines

In creating HDL code for both the top-level design and individual modules, it is extremely important to follow "good" coding practices. Following are general guidelines:

- Ensure that your design is fully synchronous
- Use realistic timing requirements (period constraints for entire design)
- Register all module outputs

Top-Level Design Coding Guidelines

The top-level design should include all global logic, all design modules instantiated as "black boxes," and the signals that connect modules to each other and to I/O ports. Each module should be instantiated as a "black box," with only ports and port directions. Following are "good" coding practices specific to the top-level design:

- Declare all ports at the top-level
- Use meaningful signal names to connect to module ports or between modules

  Note Using the same name for the signal and its associated port aids in simulation, because top-level signal names are used during back-annotated simulation.

- Include a minimum number of modules

In addition to these practices, you *must* do the following when creating the top-level design:

- Instantiate each module as a “black box”
- Include “black box” definitions of the lower-level modules in the top-level file to determine port direction and bus width
    - ♦ VHDL Notes

      Synthesis requires component declarations for all instantiated components in the HDL code.The component can be declared in the code or in a library package included in the HDL.
    - ♦ Verilog Notes

      Synthesis requires declarations for user modules only, not library primitives. If user modules are defined and described in the same project, module declarations are unnecessary. For example, module declarations are unnecessary if your synthesis tool produces multiple EDIF netlists from a single project. However, if a user module is described in a different project, or if it is a CORE Generator module, then a module declaration is required. All port directions *must* be declared with explicit statements in the module definition.
- Infer the following resources:
    - ♦ All I/O registers
    - ♦ 3-state buffers that drive the same net or bus

  Note If bidirectional signals are outputs of a lower-level module, declare them in the HDL code as “inout” signal types in both the top-level component declaration and the module-level port map.

Note You cannot include a module inside of another module. In addition, multiple instantiations of the same module are not directly supported. Each module instantiation must have a separate module definition, even if module instantiations will use the same port definitions and functions.

Module Coding Guidelines

Unlike top-level design entry, design entry for individual modules can occur after the Initial Budgeting phase and even at the same time as another module's Active Module Implementation phase.

In creating HDL code for the individual modules, adhere to the following recommendations to make your design implementation go smoothly:

- Write individual modules as independent designs

  Self-contained modules with minimal dependence on outside resources are implemented optimally. Following are examples of how to achieve independence:

  - ♦ Use the minimum number of ports on each module
  - ♦ Do not design modules to be dependent on chip resources with specific locations

    For example, a module should not require that a BRAM be located in the column adjacent to the module.
  - ♦ Include minimal global logic

    Examples of global logic are I/O pins leading onto or off of the chip, DLLs, or global clock resources.
- Define ports exactly as they appear in the top-level design

  Ports are connections in and out of a module that are generally connected to a wire or signals in the top-level design.

Synthesizing your Designs

Synthesize your HDL files as described in the documentation for your synthesis tool. You must create a separate netlist file for each of the modules and for the top-level design. Synthesis of a Modular Design requires the following special considerations:

- Each module in the design must have a unique netlist.

  Most synthesis tools generate only one netlist for each project. To meet the "separate netlist" requirement of Modular Design, you must synthesize lower-level modules separately from the top-level design, and you must create a separate project for each module as well as for the top-level design.

  Note LeonardoSpectrum allows you to create multiple EDIF netlists from a single project. See the "Creating a Netlist for Each Module (LeonardoSpectrum)" section for more information.

- In each module design, disable settings that insert I/O pads.

  See the "Vendor Specific Notes for Synthesis" section for information on how to disable this setting for your synthesis tool.

- In the top-level design, enable settings that insert I/O pads.
- In the top-level design, the names of lower-level modules must be identical to their file names.

  If these names do not match, NGDBuild cannot match the module names specified in the top-level netlist to the module netlists.

- In the top-level design, synthesize all lower-level modules as black boxes.

  Black box instantiation may require the use of a synthesis tool directive. If lower-level modules are not synthesized as black boxes, NGDBuild outputs an error during the Initial Budgeting phase.

Running Initial Budgeting

During the Initial Budgeting phase, the team leader assigns top-level constraints to the design. Both the top-level timing constraints and the area constraints for each module must be defined.

1. Change directories to your top-level design directory inside your “implementation” directory.

2. Translate your top-level netlist file into a Xilinx file format using the following command. You can use either an EDIF netlist or an NGC netlist from XST. If you use an NGC file as your top-level design, be sure to specify the .ngc extension as part of your design name.

   ```
   ngdbuild -modular initial design_name
   ```

   Module files must not be included in the top-level directory. At this point, the modules instantiated in the top-level design must be represented as unexpanded blocks in the resulting NGD file.

   Note In this step, NGDBuild produces two files, *design_name*.ngd and *design_name*.ngo. The *design_name*.ngo file is used during subsequent Modular Design steps, while *design_name*.ngd is not.

3. Apply top-level constraints, such as clock periods, using the Constraints Editor. Use the following command to invoke the Constraints Editor:

   ```
   constraints_editor design_name.ngd
   ```

   Note If a clock net in the top-level design does not have a clock load, the clock does not appear in the Constraints Editor. You must enter the timing constraints manually in the UCF file.

   Refer to the Constraints Editor online Help for details about commands and settings. Also refer to the *Constraints Guide* for information on constraints.

4. Select **File** → **Save** to save your UCF file and then close the Constraints Editor.

5. Use the following command to invoke the Floorplanner:

   ```
   floorplanner design_name.ngd
   ```

6. Select **File** → **Read Constraints** to read in the UCF file you modified in the Constraints Editor.

7. Apply location constraints, including constraints for the following:

   ♦ Top-level I/O ports

     In the Design Hierarchy window, expand the Primitives icon. Drag the port to the Floorplan window.

   ♦ Top-level logic, such as global buffers, 3-state buffers, flip-flops, and look-up tables

     In the Design Hierarchy window, expand the Primitives icon. Drag the primitive to the Floorplan window.

     Note BUFTs require some special considerations. See the "Initial Budgeting Phase" section for details.

   ♦ "Pseudo logic" that represents module ports

     Before assigning area constraints for each module, make sure that autofloorplanning is enabled. Select **Floorplan** → **Distribute Options**. In the Distribution Options dialog box, make sure **Autofloorplan as needed** is selected. When you assign an area constraint for a module, the Floorplanner positions the pseudo logic automatically.

     To reposition a port manually, select the port in the Floorplan window and drag it to its desired location. For best results, place the ports just outside the defined area for the module.

     Note Manually positioned ports are marked as unavailable for automatic floorplanning. To revert back to automatic floorplanning, delete the manually placed ports from the floorplanned design. The ports are automatically positioned the next time you resize or move an assigned area, assuming the **Autofloorplan as needed** option is selected in the Distribution Options dialog box. If this option is *not* selected, you can select the **All Ports** option in the Distribution Options dialog box and click the **Floorplan** button to place the ports automatically.

   ♦ Each module

     In the Design Hierarchy window, select the module and use the **Floorplan** → **Assign Area Constraint** command to constrain the module to an area.

Refer to the Floorplanner online Help for details about commands and settings.

8. Select **`File`** → **`Write Constraints`** to write out the UCF file.
9. Close the Floorplanner and save the FNF file.

Implementing an Active Module

During this phase, the team members implement the top-level design with only the "active" module expanded. "Active" refers to the module on which you are currently working. The full suite of Xilinx implementation tools is available for this phase. You can use any MAP or PAR command line options as well as the Constraints Editor and Floorplanner. However, PAR's re-entrant routing feature is not supported. If you use the FPGA Editor, be sure to leave area constraints and placement information from previous steps intact.

You must perform the following steps for each module. Team members can implement their modules in parallel.

Note You *cannot* use NCD files from previous software releases with Modular Design in this release. You must generate new NCD files with the current release of the software.

1. Copy the following files to the local module directory in which you will implement the module:

   Note Xilinx recommends keeping a separate directory for the files you synthesize and the files you implement.

   - Synthesized module netlist file (for example, *module_name*.edn or *module_name*.ngc)
   - UCF file the team leader created in the Initial Budgeting phase (from the top-level directory in the "implementation" directory). Rename this file from *design_name*.ucf to *module _name*.ucf.

     Note Copying the UCF file ensures that each module is implemented with a consistent set of timing and placement constraints. It also allows you to add module-specific constraints to the local copy of the UCF file as needed.

2. Change directories to the local module directory.

3. Run NGDBuild in active module mode as follows:

   ```
   ngdbuild -uc module_name.ucf -modular module
   -active module_name
   top_level_design_directory_path/design_name.ngo
   ```

   The output NGD file is named after the top-level design and contains implementation information for both the top-level design and the individual module.

4. If necessary, create module level timing constraints using the Constraints Editor as follows:

   a) Use the following command to invoke the Constraints Editor:

   ```
   constraints_editor design_name.ngd
   ```

   b) In the New dialog box, select the *module_name*.ucf file and click **OK**.

   c) Modify the constraints. Do *not* modify the timing or placement constraints entered in the original top-level UCF file.

   Note If you define an OFFSET constraint relative to a module port, a TPSYNC constraint is automatically created for that port net. The path from the synchronous element within the module to the module port is analyzed to create offset timing. Offset timing does not include the clock delay to the synchronous element within the module.

   d) Select **File** → **Save** to save your UCF file and then close the Constraints Editor.

   Refer to the Constraints Editor online Help for details about commands and settings. Also refer to the *Constraints Guide* for information on constraints.

5. Annotate the constraints from the local UCF file to the module using the following command. The –uc option ensures that the constraints from the local UCF file are annotated.

   ```
   ngdbuild -uc module_name.ucf -modular module
   -active module_name
   top_level_directory_path\design_name.ngo
   ```

6. Map the module using the following command. In this step, you are mapping the logic of the design with only the active module expanded.

   ```
   map design_name.ngd
   ```

   No command line options are required, because all the modular design information is encoded in the input NGD file.

   Note MAP expands module range constraints as a range for each slice within the module and writes this information to the output PCF file.

7. Place and route the module using the following command. In this step, you are placing and routing the logic of the design with only the active module expanded.

   ```
   par -w design_name.ncd design_name_routed.ncd
   ```

   The "_routed" syntax ensures that you do not overwrite your mapped design. The –w option ensures that any previous versions of *design_name*_routed.ncd are overwritten. However, you can use any syntax you prefer. No command line options are required, because all the modular design information is encoded in the input NCD file.

   Note If the area specified for the module cannot contain the physical logic for the module because it is sized incorrectly, you must regenerate the UCF file generated during the Initial Budgeting phase, and you must run the entire Initial Budgeting phase again.

8. Run TRACE on the implemented design to check the timing report (TWR or TWX file) for timing issues. Verify that your top-level timing constraints are met.

   ```
   trce design_name_routed.ncd
   ```

   Note By default, a summary report is generated. You can also choose to generate an error or verbose report. See the "TRACE" chapter for details.

9. If necessary, use the Floorplanner to reposition logic as follows:

   Note The Floorplanner should only be used if the module implementation is unsatisfactory, for example, if it does not meet timing constraints.

   a) Use the following command to invoke the Floorplanner:

      ```
      floorplannerdesign_name.ncd
      ```

   b) Reposition the logic.

   c) Map, place, and route your design again, as described in steps 6 and 7.

      Note When mapping your design, you must use the MAP –fp option to ensure that your updated MFP file is used.

10. Publish the implemented module file to the centrally located PIMs directory set up by the team leader:

    ```
    pimcreate pim_directory_path -ncd
    design_name_routed.ncd
    ```

    This command creates the appropriate module directory inside the PIMs directory that you specify. It then copies the local, implemented module files, including the NGO, NGM and NCD files, to the module directory inside the PIMs directory and renames the NCD and NGM files to *module_name*.ncd and *module_name*.ngm. The –ncd option specifies the fully routed NCD file that should be published.

Note You can simulate the module after running MAP or PAR as described in the "Simulating an Active Module" section.

Assembling the Modules

This is the final phase of Modular Design, in which the team leader assembles the previously implemented modules into one design. You use the physically implemented modules located in the PIMs directory as well as the top-level design file in the top-level directory to accomplish this.

1. Change directories to your top-level design directory in your “implementation” directory.

2. To incorporate all the logic for each module into the top-level design, run NGDBuild as follows:

   ```
   ngdbuild -modular assemble -pimpath
   pim_directory_path design_name.ngo
   ```

   NGDBuild generates an NGD file from the top-level UCF file, the top-level design’s NGO file, and each PIM’s NGO file.

   Note Because you are using all of the PIMs published to the PIMs directory, you do *not* need to specify the –use_pim option. If you want to use only some of the PIMs in the PIMs directory, do not run the standard Modular Design flow. Instead, see the “Running the Sequential Modular Design Flow” section.

3. Map the logic of the full design as follows:

   ```
   map design_name.ngd
   ```

   No command line options are required, because all the modular design information is encoded in the input NGD file. MAP uses the NCD and NGM files from each of the module directories inside the PIMs directory to accurately recreate the module logic.

4. Place and route the logic of the full design as follows:

   ```
   par -w design_name.ncd design_name_routed.ncd
   ```

   No command line options are required, because all the modular design information is encoded in the input NCD file. PAR uses the NCD file from each of the module directories inside the PIMs directory to accurately reimplement the module logic.

5. Run TRACE on the implemented design to check the timing report (TWR or TWX file) for timing issues. Verify that your top-level timing constraints are met.

   ```
   trce design_name_routed.ncd
   ```

Note By default, a summary report is generated. You can also choose to generate an error or verbose report. See the "TRACE" chapter for details.

6. If desired, simulate the design as follows:

   a) Run NGDAnno as follows:

      ngdanno -o *design_name*.**nga** *design_name*.**ncd** *design_name*.**ngm**

      Note Using an NGM file is optional but recommended. It provides valuable information, such as a record of the design hierarchy including internal signal and instance names, that may not be preserved in the NCD file.

   b) Run one of the following commands to create a netlist that can be simulated:

      ngd2vhdl *design_name*.**nga**

      ngd2ver *design_name*.**nga**

   c) Use a simulator to simulate the netlist.

Simulating an Active Module

In addition to simulating the final design, you can simulate the active module (*design_name*.ncd) after running MAP or PAR. Following are the two simulation methods available during the Active Module Implementation phase. Each has its advantages and disadvantages.

- Simulation with the top-level design as context

  With this method, you back-annotate and completely simulate the top-level design.

  - Advantage: The logic in the top-level design is included in the simulation.

  - Disadvantage: Inactive modules are undefined and signals connected to module ports are left dangling. As a result, you must probe and stimulate these signals to obtain meaningful simulation results.

  Note If you use VHDL, internal signals cannot be driven from the testbench, but some simulation tools allow access to these signals through a GUI or command line tool.

- Independent module simulation

  With this method, you simulate the module independent of the design context. The simulation netlist contains only module-level logic and ports and can be instantiated in a testbench that exercises just the module.

  - ♦ Advantage: You can see exactly how the module behaves, independent of the top-level design. You do not need to provide stimuli for dangling signals as you do when simulating with the top-level design as context. In addition, you can use module-level testbench files with the resulting timing simulation netlist.
  - ♦ Disadvantage: Because port loads and drivers are unknown, you must ignore delay and timing values of module ports until you can perform a complete design simulation. In addition, all ports and internal signal names appear in the back-annotated netlist in terms of the top-level netlist. The ports are named after the top-level signals to which they connect, and the internal signals are preceded with the instance name.

Running Simulation with Top-Level Design as Context

To run this type of simulation, do the following:

1. Run NGDAnno as follows:

   ngdanno -o *module_name*.**nga** *design_name*.**ncd**
   design_name.**ngm**

   Note Using an NGM file is optional but recommended. It provides valuable information, such as a record of the design hierarchy including internal signal and instance names, that may not be preserved in the NCD file.

2. Run one of the following commands to create a netlist to simulate:

   ngd2vhdl *module_name*.**nga**

   ngd2ver *module_name*.**nga**

3. Use a simulator to simulate the netlist.

Running Independent Module Simulation

To run this type of simulation, do the following:

1. Run NGDAnno as follows:

   ngdanno -o *module_name***.nga -module** *design_name***.ncd** *design_name***.ngm**

   Note Using an NGM file is optional but recommended. It provides valuable information, such as a record of the design hierarchy including internal signal and instance names, that may not be preserved in the NCD file.

2. Run one of the following commands to create a netlist to simulate:

   ngd2vhdl *module_name***.nga**

   ngd2ver *module_name***.nga**

3. Use a simulator to simulate the netlist.

Running the Sequential Modular Design Flow

If you are comfortable running the standard Modular Design flow, you can also run this flow "sequentially." This means taking information gained from one or a few module implementations and using it to improve other module implementations and your final design. The two sequential flows and their advantages are described in the following sections.

Running the Partial Design Assembly Flow

This Modular Design flow allows the team leader to run the Final Assembly phase with only some of the modules implemented. This allows you to check your partially assembled design for timing problems before all modules are completed. You can analyze the timing budget for the following:

- Nets that connect implemented and unimplemented modules
- Nets that connect the implemented modules to one another

The Initial Budgeting and Active Module Implementation phases are the same as in the standard flow. The Final Assembly phase differs slightly, as shown in the following figure.

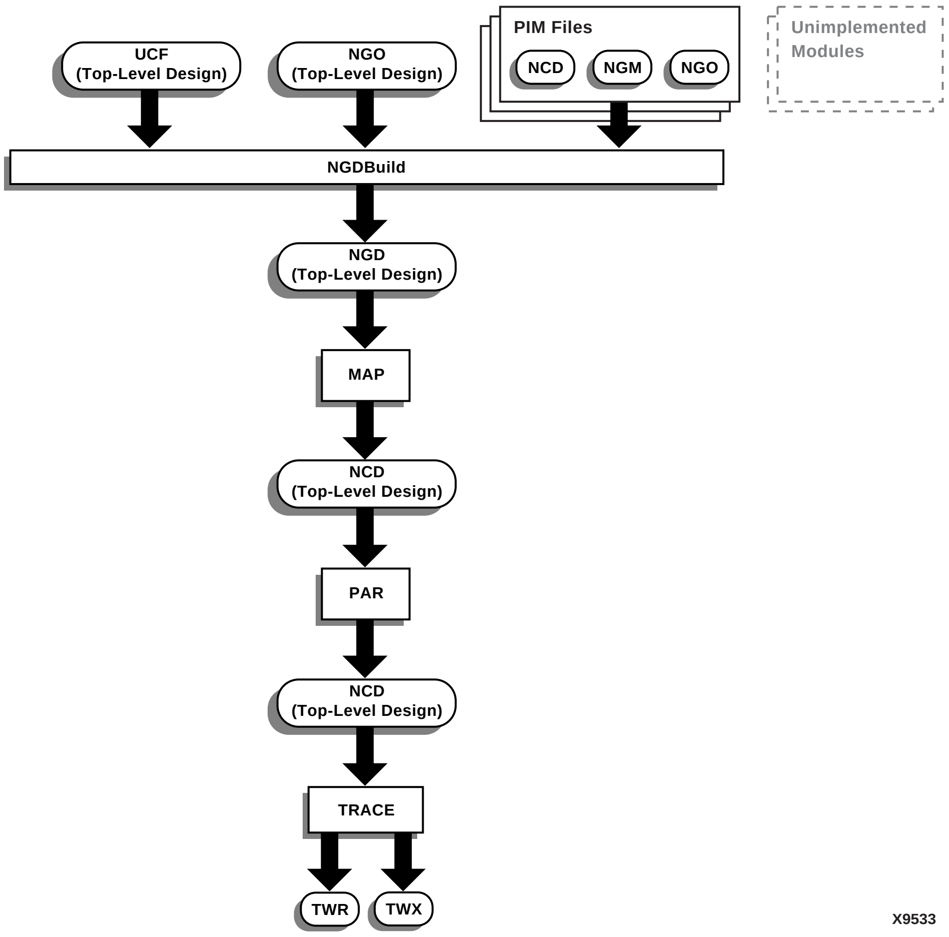


Figure 3-8 Final Assembly Phase for Partial Design Assembly Flow

Run the Partial Design Assembly flow as follows:

1. Enter your design using the guidelines described in the "Entering the Design" section.

2. Synthesize your HDL files as described in the documentation for your synthesis tool. You must create a separate netlist file for each of the modules as well as the top-level design. For guidelines, see the "Synthesizing your Designs" section.

   Note For modules, disable settings that insert I/O pads.

3. Run Initial Budgeting for your design as described in the "Running Initial Budgeting" section.

4. Make sure the appropriate modules have been implemented as described in the "Implementing an Active Module" section.

5. Change directories to the top-level design directory in your "implementation" directory to begin the Final Assembly phase.

6. To incorporate the logic for the specified modules into the top-level design, run NGDBuild as follows.

   ```
   ngdbuild -u -modular assemble -pimpath
   pim_directory_path -use_pim module_name1 -use_pim
   module_name2... design_name
   ```

   The –u option instructs NGDBuild to ignore the modules that are missing from the PIMs directory. See the "–u (Allow Unexpanded Blocks)" section and the "–modular assemble (Module Assembly)" section of the "NGDBuild" chapter for information on the NGDBuild options.

   Note Use the –use_pim option to specify *only* the modules that were published to the PIMs directory. You must use the *same* naming conventions used during the Active Module Implementation phase, including the proper capitalization.

7. Map the logic of the partially implemented design as follows:

   ```
   map design_name.ngd
   ```

   Note MAP does not trim port nets associated with unimplemented modules. The MAP report (MRP file) lists each unimplemented module and its associated untrimmed logic.

8. Place and route the logic of the partially implemented design as follows:

   ```
   par -w design_name.ncd design_name_routed.ncd
   ```

9. Run TRACE on the implemented design to check the timing report (TWR or TWX file) for timing issues. Verify that your top-level timing constraints are met.

   ```
   trce design_name_routed.ncd
   ```

   Note By default, a summary report is generated. You can also choose to generate an error or verbose report. See the "TRACE" chapter for details.

Running the Incremental Guide Flow

This flow allows the team leader to implement the top-level design with both the previously implemented module or modules and the "active" module expanded. By implementing your design this way, you avoid both global resource contention issues and the requirement to assign all pseudo logic, because each active module is aware of the resources used by previously implemented modules.

The Initial Budgeting phase is similar to that of the standard flow. The Active Module Implementation phase differs slightly, as shown in the following figure. In addition, the Final Assembly phase is not needed in this flow.

Note The NGM and NCD files from the previously implemented modules are automatically read into MAP and PAR. You do not need to specify these files at the command line.

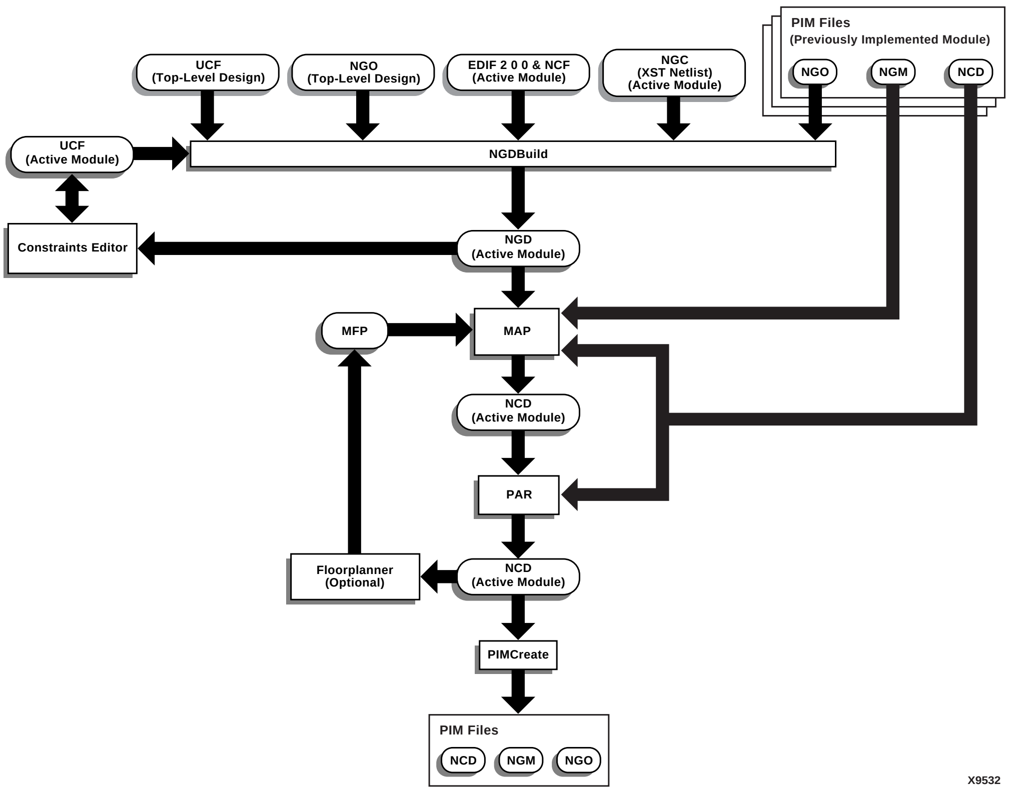


Figure 3-9 Active Module Implementation Phase for Incremental Guide Flow

Run the Incremental Guide flow as follows:

1. Enter you design using the guidelines described in the "Entering the Design" section.

2. Synthesize the HDL files as described in the documentation for your synthesis tool. You must create a separate netlist file for each of the modules as well as the top-level design. For guidelines, see the "Synthesizing your Designs" section.

   Note For modules, disable settings that insert I/O pads.

3. Run Initial Budgeting for your design as described in the "Running Initial Budgeting" section.

   Note You do *not* need to automatically position pseudo logic for the entire design, only pseudo logic associated with the first module. Using the Incremental Guide flow, the logic from the previously implemented module or modules is used rather than the pseudo logic created for the top-level design.

4. In your "implementation" directory, create a directory for each module to be implemented. These directories are different from those in the PIMs directory.

5. Implement the first module as described in the "Implementing an Active Module" section.

6. To implement the next module, copy the following files to the appropriate module directory in your "implementation" directory:

   ♦ Synthesized module netlist file (for example, *module_name*.edn or *module_name*.ngc).

   ♦ UCF file you created in the Initial Budgeting phase (from the top-level directory in the "implementation" directory). Rename this file from *design_name*.ucf to *module _name*.ucf.

   Note Copying the UCF file ensures that each module is implemented with a consistent set of timing and placement constraints. It also allows you to add module-specific constraints to the local copy of the UCF file as needed.

7. Change directories to the appropriate module directory.

8. Run NGDBuild as follows. During this step, the top-level design is implemented with both the previously implemented module or modules and the active module expanded.

   ```
   ngdbuild -uc design_name.ucf -modular module
   -active module_name -pimpath pim_directory_path
   -use_pim module_name1 -use_pim module_name2
   top_level_directory_path\design_name.ngo
   ```

   Note Use the –use_pim option to specify *only* the modules that were published to the PIMs directory. You must specify *all* published modules each time you run this command. You must use the *same* naming conventions used during the Active Module Implementation phase, including the proper capitalization.

9. If necessary, create module level timing constraints using the Constraints Editor as follows:

   a) Use the following command to invoke the Constraints Editor:

      ```
      constraints_editor design_name.ngd
      ```

   b) In the New dialog box, select the *module_name*.ucf file and click **OK**.

   c) Modify the constraints.

      Note If you define an OFFSET constraint relative to a module port, a TPSYNC constraint is automatically created for that port net. The path from the synchronous element within the module to the module port is analyzed to create offset timing. Offset timing does not include the clock delay to the synchronous element within the module.

   d) Select **File** → **Save** to save your UCF file and then close the Constraints Editor.

   Refer to the Constraints Editor online Help for details about commands and settings. Also refer to the *Constraints Guide* for information on constraints.

10. Annotate the constraints from the local UCF file to the module using the following command. The –uc option ensures that the constraints from the local UCF file are annotated.

    ```
    ngdbuild -uc module_name.ucf -modular module
    -active module_name -pimpath pim_directory_path
    ```

```
-use_pim module_name1
top_level_directory_path\design_name.ngo
```

11. Map the module using the following command. In this step, you are mapping the logic of the design with both the “active” module and the previously implemented module or modules expanded.

    ```
    map design_name.ngd
    ```

12. Place and route the module using the following command. In this step, you are placing and routing the logic of the design with both the “active” module and the previously implemented module or modules expanded.

    ```
    par -w design_name.ncd design_name_routed.ncd
    ```

    Note The “_routed” syntax ensures that you do not overwrite your mapped design. However, you can use any syntax you prefer. The –w options ensures that any previous versions of *design_name*_routed.ncd are overwritten.

13. Publish the implemented module file to a centrally located PIMs directory.

    ```
    pimcreate -ncd design_name_routed.ncd
    pim_directory_path
    ```

    This command creates the appropriate module directory inside the PIMs directory that you specify. It then copies the local, implemented module files, including the NGO, NGM and NCD files, to the module directory inside the PIMs directory and renames the NCD and NGM files to *module_name*.ncd and *module_name*.ngm.

14. Repeat steps 6 through 13 for each succeeding module.

There is no need to run Modular Design in Final Assembly Mode. When you place and route your last module, your resulting NCD file is your final design file.

Note If any of the constraints you alter affect your previously implemented modules, you *must* re-implement the previously implemented modules and also your active module. To do this, start with step 5 of this procedure. If your constraints only affect your active module, you do *not* need to re-implement your previously implemented modules.

Modular Design Tips

Following are tips for working with Modular Design. For additional help, use the resources at http://support.xilinx.com.

Constraints

Use the following constraints to improve your Modular Design results. For more information, see the *Constraints Guide*.

- AREA_GROUP
- COMPGRP
- FROM-TO
- LOC (on module ports using PIN statements)
- NET TPSYNC
- OFFSET (on module ports through TPSYNC groups)
- PERIOD
- PIN
- RESERVED_SITES

  Note This constraint is not supported for TBUFs, slices, multipliers, block RAMs, or CLBs.
- ROUTE
- TIG

Note Some of these constraints are automatically generated when you use the Floorplanner and Constraints Editor GUIs with your design. It is not necessary to manually add them to your design.

When assembling PIMs, whether for a partial or standard design, or when running the Incremental Guide flow, the implementation tools generally guide and implement each module identically to its individual module implementation. However, conflicts may occur that cause PAR to reroute the initial routing. Properly constraining the paths within the modules ensures that any rerouting is done correctly. Following are some general guidelines:

- Remove constraints on module ports before the Final Assembly phase.

  For example, remove any PIN LOC, TPSYNC / FROM-TO, or TPSYNC / OFFSET constraints on module ports. When the module is used as a PIM, the tools may improperly constrain or overconstrain the design.

- Store all constraints in the UCF file, not the NCF or netlist file.

  When the module is used as a PIM, NGDBuild discards timing constraints found in the NCF or netlist file.

- Place OFFSET constraints on module ports relative to the actual clock pad net, not to the module clock port.

  Note This is a current limitation with Modular Design.

- Before assembling modules or using a PIM for the Incremental Guide flow, copy the relevant constraints from the module's UCF file to the top-level UCF file.

  Copy only those constraints that apply to paths *completely* contained within the module. This ensures that all relevant constraints are considered during routing. Following are examples of constraints you should copy:

  - FROM-TO specifications (and associated TNM, TNM_NET, and TIMEGROUP definitions) for which both endpoints are within the module
  - PERIOD specifications (and associated TNM, TNM_NET, and TIMEGROUP definitions) for clocks that are used only within the module

    Note A PERIOD specification on a top-level clock controls paths inside any PIM that is clocked by that signal, so it is not necessary to replicate the specification.
  - TIG directives on nets or pins within the module, provided that the TIG is global (that is, it has no value) or applies to a specification (TSid) that has also been copied

Note You can enter global and top-level constraints with the synthesis tools, but most module-specific constraints must be entered manually or using the Constraints Editor or Floorplanner. See the online Help available from each of these GUIs for more information.

Design Size and Performance

To take full advantage of Modular Design, use this design flow with large designs that have been created with Modular Design in mind. When working with very large designs the issues of memory usage, run time, and sheer complexity often make implementation difficult. Because Modular Design allows multiple designers to work in parallel and make changes to previously implemented modules in an assembled design, overall efficiency is improved when compared with implementing a large, flat design. These advantages include reduced run time overall and efficient use of resources.

Note Modular Design can be used on small designs to learn Modular Design techniques, but many of the steps may be burdensome and complex for a small design.

PCI Logic

For information on using PCI logic in a modular design, see the solution record at: http://support.xilinx.com/techdocs/10521.htm.

MAP Report

After the Active Module Implementation and Final Assembly phases, review the following sections of the MAP report (MRP file) to help improve your area groups in the top-level floorplan:

- Modular Design Summary

  You can use this section to help you do the following: determine which components of your design are part of a module, distinguish whether you are running a partial or full Modular Design Assembly, and verify your design.

- Guide Report

  If you mapped your design using a guide file, you can use this section to find out the guide mode used (EXACT or LEVERAGE) and the percentage of objects that were successfully guided.

- Area Group Summary

  You can use this section to find out the results for each area group. MAP uses area groups to specify a group of logical blocks that are packed into separate physical areas.

Note If you want to debug NOMERGE errors and warnings after the Active Module Implementation phase, set the XIL_MAP_LISTPORTNETS environment variable. When you set this environment variable, the Modular Design Summary section of the MRP report includes a list of the port nets for the active modules.

PAR Reports

After the Final Assembly phase, review the “Guide Summary Report” section of the PAR report file (*design_name*.par). This section provides a summary of how many components and signals in the design were guided by the files in the PIMs directory.

For a more detailed report, review the Guide Report File (*design_name*.grf). This file includes the same “Guide Summary Report” section as the PAR file and also includes a “Detail Report” section. The “Detail Report” section contains a section for each PIM guide file. Each of these sections contains the following information about the components and signals that PAR guided or attempted to guide in the final design:

- Guided comps located in the guided site

  This section lists the components that were matched and guided from the PIM to the final design. It also lists the component’s location on the chip.

- Guided comps unable to be located in the guided site

  This section lists the components that were matched but could not be guided in the final design based on their location in the PIM. It also lists the component’s location on the chip.

- Components in the Guide File that did not match Placement

  This section lists the components in the PIM that could not be matched in the final design.

- Signals in the Guide File that did not match

  This section lists the signals in the PIM that could not be matched in the final design.

XFLOW Automation of Modular Design

You can use Xilinx’s XFLOW command line tool to automate the Modular Design flow. See the “XFLOW” chapter for general information on this tool. For information on the flow types specific to Modular Design, see the following sections:

- “–initial (Initial Budgeting of Modular Design)”
- “–module (Active Module Implementation)”
- “–assemble (Module Assembly)”

Modular Design Troubleshooting

Following are troubleshooting tips for working with Modular Design. For additional troubleshooting help, use the resources at http://support.xilinx.com.

Multiple Output Ports MAP Error

If you include a signal in a module that drives more than one output port, you receive a MAP error. For example, in the following Verilog code, ctrl is a 4 bit bus that includes bits a_out and b_out. Because a_out and b_out drive two output ports but are the same signal, MAP generates an error when processing the design.

```
output [3:0] ctrl;
.
.
.
wire a_out = ~flag_1;
wire b_out = ~flag_1;
.
.
.
assign ctrl = {1’b0, a_out, b_out, other_signal};
```

To fix issues such as these, do one of the following. The first option is recommended.

- Consolidate the ports into one port. Change the top-level design and module source code. After changing the code, you must rerun the Initial Budgeting phase for the top-level design and rerun the Active Module Implementation phase for each module.
- Add buffers to create one signal for each port in the module source code. The following figure shows an example.

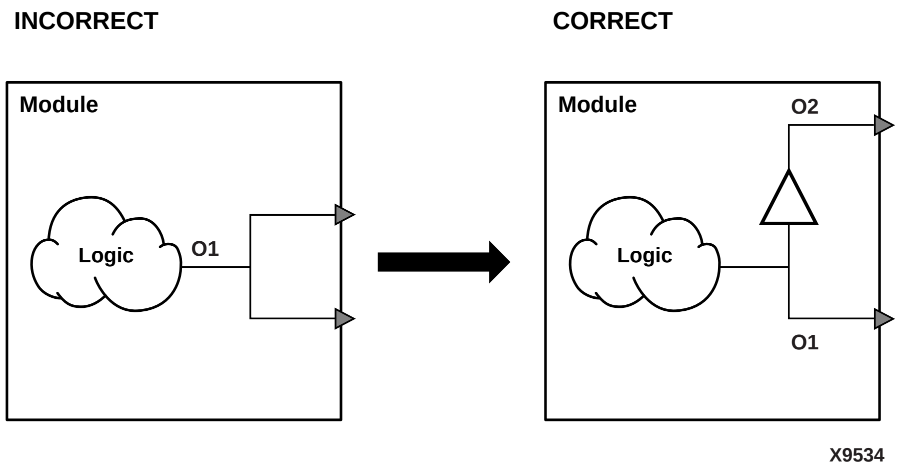


Figure 3-10 Module with Added Buffer

- Replicate logic so there is one signal for each port in the module source code. The following figure shows an example.

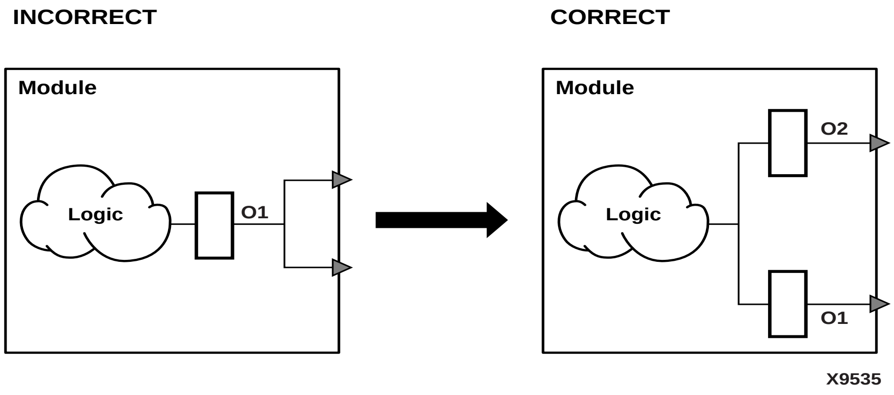


Figure 3-11 Module with Replicated Logic

Part Type Specification

If you are targeting a part different from the one specified in your source design, you must specify the part type using the -p option *every time* you run NGDBuild. The syntax for the –p option is described in the "–p (Part Number)" section of the "Introduction" chapter.

Constraints Not Working in Active Module Implementation

If it appears that a constraint is not being processed, make sure there are not multiple versions of the same constraint defined in the NCF or UCF file. In most cases, if a constraint is defined multiple times, the last definition of the constraint overwrites any previous definitions. To specify more than one value for a particular constraint, list all the values on the same line.

Following are examples of correct and incorrect syntax:

- Correct

  The following syntax reserves both site “PAD43” and site “PAD21” for the module named “A”:

  ```
  MODULE A RESERVED_SITES = “PAD43, PAD21”;
  ```

- Incorrect

  In the following syntax, the second RESERVED_SITE constraint overwrites the first, and the site “PAD43” is not reserved:

  ```
  MODULE A RESERVED_SITES = “PAD43”;
  MODULE A RESERVED_SITES = “PAD21”;
  ```

Resource Contention or Timing Constraints Not Met in Final Assembly

Resource contention among modules can occur due to module use of global logic or routing resources. Also, even if each module meets its timing constraints, the overall design may not meet its timing constraints due to additive delays. If either of these conditions occur, reimplement one or more modules as described in the “Implementing an Active Module” section before proceeding to the Final Assembly phase.

Note Although it is possible to use tools during the Final Assembly phase to directly manipulate resources contained in a module, this is not recommended, because it renders published module information invalid for future iterations.

Windows 98 Command Line Limitation

The Windows 98 platform limits the number of characters that can be entered on the command line. If your command exceeds this limit, place your command in an SCR or BAT file to execute it.

Vendor Specific Notes for Synthesis

Use the following procedures for your particular synthesis tool. If your tool is not listed, refer to your tool's user documentation.

Synplify or FPGA Express/FPGA Compiler II, version 3.3.1 or earlier

Use the following procedures if you are synthesizing with Synplify or FPGA Express/FPGA Compiler II (version 3.3.1 or earlier).

Creating a Netlist for Each Module (Synplify or FPGA Express/FPGA Compiler II, version 3.3.1 or earlier)

Each design project creates one netlist. To create a netlist for each module, do the following:

1. Create a project for the top-level design and for each lower-level module.
2. Synthesize the top-level project with I/O insertion and the lower-level modules without I/O insertion.

Disabling I/O Insertion for a Module (Synplify or FPGA Express/FPGA Compiler II, version 3.3.1 or earlier)

To disable I/O insertion for a module, do the following:

1. Select **`Target`** → **`Set Device Options`**.
2. In the Set Device Options dialog box, select **`Disable I/O Insertion`**.

Instantiating Primitives (Synplify or FPGA Express/ FPGA Compiler II, version 3.3.1 or earlier)

For both VHDL and Verilog, you do not need to declare modules when calling primitives and mapping ports. Synplify provides Virtex primitives in the following areas:

- VHDL: “library virtex” located in $SYNPLICITY/lib/xilinx
- Verilog: and “virtex.v”located in $SYNPLICITY/lib/xilinx

FPGA Express/FPGA Compiler II, version 3.4 or later

Use the following procedures if you are synthesizing with FPGA Express/FPGA Compiler II (version 3.4 or later).

Creating a Netlist for Each Module (FPGA Express/ FPGA Compiler II, version 3.4 or later)

Use the Incremental Synthesis feature to synthesize each design module individually within a project. To create a netlist for each module, do the following:

1. In the FPGA Express/FPGA Compiler II Constraints Editor, select the Modules tab.
2. In the Block Partition column, set the Block Root attribute for the top-level design and for each module listed.
3. Export the design to produce a separate EDIF file for each module.

Disabling I/O Insertion for a Module (FPGA Express/ FPGA Compiler II, version 3.4 or later)

To disable I/O insertion for a module, do the following:

1. Select **`Synthesis`** → **`Update`**.
2. Select **`Synthesis`** → **`Create Implementation`**.
3. In the Create Implementation dialog box, select **`Do not insert I/O pads`**.

Instantiating Primitives (FPGA Express/FPGA Compiler II, version 3.4 or later)

Instantiating primitives differs based on whether you use VHDL or Verilog.

- VHDL: You must declare all instantiated components in the VHDL.
- Verilog: You do not need to declare modules in the Verilog code.

  Note If you instantiate an IBUFG in the top-level design code, FPGA Express inserts IBUF before IBUFG, which causes an NGDBuild error. To avoid this, instantiate the IPAD, omitting the port declaration.

LeonardoSpectrum

Use the following procedures if you are synthesizing with LeonardoSpectrum.

Creating a Netlist for Each Module (LeonardoSpectrum)

To create a netlist for each module, you can create multiple netlists from a single project using the GUI or using a script.

Following is a script example for a VHDL design:

```
set part v50ecs144
load_library xcve
read ./top.vhd
optimize -target xcve -hier preserve
present_design .work.top.modular
auto_write -format edf top.edf
read ./module_a.vhd
read ./module_b.vhd
read ./module_c.vhd
optimize -target xcve -hier preserve
present_design .work.module_a.modular
auto_write -format edf module_a.edf
present_design .work.module_b.modular
auto_write -format edf module_b.edf
present_design .work.module_c.modular
auto_write -format edf module_c.edf
```

Following is a script example for a Verilog design:

```
set part v50ecs144
load_library xcve
read ./module_a.v
read ./module_b.v
read ./module_c.v
read ./top.v
optimize -target xcve -hier preserve
present_design .work.module_a.INTERFACE
auto_write -format edf module_a.edf
present_design .work.module_b.INTERFACE
auto_write -format edf module_b.edf
present_design .work.module_c.INTERFACE
auto_write -format edf module_c.edf
NOOPT .work.module_a.INTERFACE
NOOPT .work.module_b.INTERFACE
NOOPT .work.module_c.INTERFACE
present_design .work.top.INTERFACE
auto_write -format edf top.edf
```

Disabling I/O Insertion for a Module (LeonardoSpectrum)

To disable I/O insertion for a module, do the following:

1. Select the Quick Setup tab.
2. Make sure the **`Insert I/O Pads`** checkbox is *deselected*.

Instantiating Primitives (LeonardoSpectrum)

Instantiating primitives differs based on whether you use VHDL or Verilog.

- VHDL: You must declare all instantiated components in the code.
- Verilog: You do not need to declare modules in the code.

XST

Use the following procedures if you are synthesizing with Xilinx Synthesis Technology (XST). See the *Xilinx Synthesis Technology (XST) User Guide* for more information.

Creating a Netlist for Each Module (XST)

Use the Incremental Synthesis feature to synthesize each design module individually within a project. To create a netlist for each module, do the following:

1. In the Project Navigator, select your module design in the Source window.
2. Select Synthesize in the Process window.
3. Select **`Process`** → **`Properties`**.
4. In the Xilinx Specific Options tab of the Process Properties dialog box, make sure the **`Incremental Synthesis`** checkbox is *selected*.
5. Export the design to produce a separate EDIF file for each module.

Disabling I/O Insertion for a Module (XST)

To disable I/O insertion for a module, do the following:

1. In the Xilinx Project Navigator, select your module design in the Source window.
2. Select Synthesize in the Process window.
3. Select **`Process`** → **`Properties`**.
4. In the Xilinx Specific Options tab of the Process Properties dialog box, make sure the **`Add I/O Buffers`** checkbox is *deselected*.

Instantiating Primitives (XST)

XST instantiates primitives automatically.

HDL Code Examples

Following are code examples for your reference.

Top-Level Design

The top-level design should include all global logic, all design modules, and the logic that connects modules to each other and to I/O ports. Each module should be instantiated as a “black box,” with only ports and port directions. For general coding guidelines, see the “General Coding Guidelines” section.

VHDL Example: Top-Level Design

```
library IEEE;
use IEEE.std_logic_1164.all;
entity top is port (ipad_dll_clk_in: in std_logic;
    dll_rst : in std_logic;
    top2a_c: in std_logic;
    top2b: in std_logic;
    obuft_out: out std_logic;
mod_c_out: out std_logic;
    moda_clk_pad: in std_logic;
    moda_data: in std_logic;
    moda_out: out std_logic;
    modb_clk_pad: in std_logic;
    modb_data: in std_logic;
    modb_out: out std_logic;
    modc_clk_pad: in std_logic;
    modc_data: in std_logic;
    modc_out: out std_logic
) ;
end top;
architecture modular of top is
signal dll_clk_in : std_logic;
signal clk_top : std_logic;
signal dll_clk_out: std_logic;
signal a2top_obuft_i: std_logic;
signal a2c: std_logic;
signal a2b: std_logic;
signal b2top_obuft_t: std_logic;
signal b2c: std_logic;
signal b2a: std_logic;
signal c2and2: std_logic;
signal c2a: std_logic;
signal a_and_c: std_logic;
signal moda_clk: std_logic;
signal modb_clk: std_logic;
signal modc_clk: std_logic;
component IBUFG is port( I : in std_logic; O : out std_logic);
end component;
component CLKDLL is port (
  CLKIN : in std_logic;
  CLKFB : in std_logic;
  RST : in std_logic;
  CLK0 : out std_logic;
  CLK90 : out std_logic;
  CLK180 : out std_logic;
  CLK270 : out std_logic;
  CLKDV : out std_logic;
  CLK2X : out std_logic;
  LOCKED : out std_logic);
  end component;
  component BUFG
  port (
  I : in std_logic;
  O : out std_logic);
  end component;
component BUFGP
  port (
  I : in std_logic;
  O : out std_logic);
  end component;
-- Declare modules at top-level to get port directionality
component module_a is port( CLK_TOP: in std_logic;
    B2A_IN: in std_logic;
    TOP2A_IN: in std_logic;
    C2A_IN: in std_logic;
    MODA_DATA : in std_logic;
    MODA_CLK : in std_logic;
    A2B_OUT: out std_logic;
    A2TOP_OBUFT_I_OUT: out std_logic;
    A2c_ouT: out std_logic;
    MODA_OUT : out std_logic
);
end component;

component module_b is port( CLK_TOP: in std_logic;
    A2B_IN: in std_logic;
    TOP2B_IN: in std_logic;
    A_AND_C_IN: in std_logic;
    MODB_DATA: in std_logic;
    MODB_CLK: in std_logic;
    MODB_OUT : out std_logic;
    B2A_OUT: out std_logic;
    B2TOP_OBUFT_T_OUT: out std_logic;
    B2C_OUT: out std_logic);
end component;
component module_c is port( CLK_TOP: in std_logic;
    B2C_IN: in std_logic;
    TOP2A_C_IN: in std_logic;
    A2C_IN: in std_logic;
    MODC_DATA: in std_logic;
    MODC_CLK: in std_logic;
    MODC_OUT: out std_logic;
    C2A_OUT: out std_logic;
    C2TOP_OUT: out std_logic;
    C2AND2_OUT: out std_logic);
end component;
begin
ibuf_dll: IBUFG port map(I =>ipad_dll_clk_in,
    O => dll_clk_in);
dll_1: CLKDLL port map(CLKIN => dll_clk_in,
    CLKFB => clk_top,
    CLK0 => dll_clk_out,
    RST => dll_rst);
globalclk: BUFG port map(O => clk_top,
    I => dll_clk_out);
bufg_moda : BUFGP port map (O => moda_clk,
    I => moda_clk_pad);
bufg_modb : BUFGP port map (O => modb_clk,
    I => modb_clk_pad);
bufg_modc : BUFGP port map ( O => modc_clk,
    I => modc_clk_pad);

-- A simple piece of external logic at top level
```

```
a_and_c <= c2and2 and b2a;
-- Tri-state output
obuft_out <= a2top_obuft_i when b2top_obuft_t = ’0’ else ’Z’;
instance_a: module_a port map (CLK_TOP =>clk_top,
    TOP2A_IN =>top2a_c,
    C2A_IN =>c2a,
    B2A_IN => b2a,
    MODA_DATA => moda_data,
    MODA_CLK => moda_clk,
    MODA_OUT => moda_out,
    A2B_OUT => a2b,
    A2TOP_OBUFT_I_OUT => a2top_obuft_i,
    A2C_OUT => a2c) ;
instance_b: module_b port map ( CLK_TOP => clk_top,
    TOP2B_IN => top2b,
    A2B_IN => a2b,
    A_AND_C_IN => a_and_c,
    MODB_DATA => modb_data,
MODB_CLK => modb_clk,
MODB_OUT => modb_out,
    B2TOP_OBUFT_T_OUT => b2top_obuft_t,
    B2C_OUT => b2c,
    B2A_OUT => b2a);
instance_c: module_c port map ( CLK_TOP => clk_top,
    TOP2A_C_IN => top2a_c,
    B2C_IN => b2c,
    A2C_IN => a2c,
    MODC_DATA => modc_data,
MODC_CLK => modc_clk,
MODC_OUT => modc_out,
    C2TOP_OUT => mod_c_out,
    C2AND2_OUT => c2and2,
    C2A_OUT => c2a);
end modular;
```

Verilog Example: Top-Level Design

```
  module top (ipad_dll_clk_in, dll_rst, top2a_c, top2b, obuft_out,
mod_c_out, moda_data, moda_clk_pad, moda_out, modb_data,
modb_clk_pad, modb_out, modc_data, modc_clk_pad, modc_out) ;
input ipad_dll_clk_in;
input dll_rst;
input top2a_c;
input top2b;
output obuft_out;
output mod_c_out;
input moda_data;
input moda_clk_pad;
output moda_out;
input modb_data;
input modb_clk_pad;
output modb_out;
input modc_data;
input modc_clk_pad;
output modc_out;
//wire ipad_dll_clk_out;
wire clk_top;
wire dll_clk_out;
wire a2top_obuft_i;
wire a2c;
wire a2b;
wire b2top_obuft_t;
wire b2c;
wire b2a;
wire c2and2;
wire c2a;
wire a_and_c;
wire moda_clk;
wire modb_clk;
wire modc_clk;
IBUFG ibuf_dll (.I(ipad_dll_clk_in),
    .O(dll_clk_in));
CLKDLL dll_1 (.CLKIN(dll_clk_in),
    .CLKFB(clk_top),
    .CLK0(dll_clk_out),
    .RST(dll_rst));
BUFG globalclk (.O(clk_top),
    .I(dll_clk_out));
BUFGP bufg_moda (.O(moda_clk),
    .I(moda_clk_pad));
BUFGP bufg_modb (.O(modb_clk),
    .I(modb_clk_pad));
BUFGP bufg_modc (.O(modc_clk),
    .I(modc_clk_pad));
// A simple piece of external logic at top level
assign a_and_c = c2and2 && b2a;
// Tri-state output
assign obuft_out = (!b2top_obuft_t) ? a2top_obuft_i : 1’bz;
module_a instance_a (.CLK_TOP(clk_top),
    .B2A_IN(b2a),
    .TOP2A_IN(top2a_c),
    .C2A_IN(c2a),
    .MODA_DATA(moda_data),
    .MODA_CLK (moda_clk),
    .MODA_OUT (moda_out),
    .A2B_OUT(a2b),
    .A2TOP_OBUFT_I_OUT(a2top_obuft_i),
    .A2C_OUT(a2c)) ;
module_b instance_b ( .CLK_TOP(clk_top),
    .TOP2B_IN(top2b),
    .A2B_IN(a2b),
    .A_AND_C_IN(a_and_c),
    .MODB_DATA(modb_data),
    .MODB_CLK(modb_clk),
    .MODB_OUT(modb_out),
    .B2TOP_OBUFT_T_OUT(b2top_obuft_t),
    .B2C_OUT(b2c),
    .B2A_OUT(b2a));
module_c instance_c ( .CLK_TOP(clk_top),
    .TOP2A_C_IN(top2a_c),
    .B2C_IN(b2c),
    .A2C_IN(a2c),
    .MODC_DATA(modc_data),
    .MODC_CLK(modc_clk),
    .MODC_OUT(modc_out),
    .C2TOP_OUT(mod_c_out),
    .C2AND2_OUT(c2and2),
    .C2A_OUT(c2a));
endmodule
```

```
// Declare modules at top-level to get port directionality
module module_a ( CLK_TOP, B2A_IN, TOP2A_IN, C2A_IN, MODA_DATA,
MODA_CLK, MODA_OUT, A2B_OUT, A2TOP_OBUFT_I_OUT, A2C_OUT) ;
input CLK_TOP ;
input B2A_IN ;
input TOP2A_IN ;
input C2A_IN ;
input MODA_DATA;
input MODA_CLK;
output MODA_OUT;
output A2B_OUT ;
output A2TOP_OBUFT_I_OUT ;
output A2C_OUT ;
endmodule

module module_b ( CLK_TOP, A2B_IN, TOP2B_IN, A_AND_C_IN, MODB_DATA,
MODB_CLK, MODB_OUT, B2A_OUT, B2TOP_OBUFT_T_OUT, B2C_OUT) ;
input CLK_TOP ;
input A2B_IN ;
input TOP2B_IN ;
input A_AND_C_IN ;
input MODB_DATA;
input MODB_CLK;
output MODB_OUT;
output B2A_OUT ;
output B2TOP_OBUFT_T_OUT ;
output B2C_OUT ;
endmodule

module module_c ( CLK_TOP, B2C_IN, TOP2A_C_IN, A2C_IN, MODC_DATA,
MODC_CLK, MODC_OUT, C2A_OUT, C2TOP_OUT, C2AND2_OUT) ;
input CLK_TOP ;
input B2C_IN ;
input TOP2A_C_IN ;
input A2C_IN ;
input MODC_DATA;
input MODC_CLK;
output MODC_OUT;
output C2A_OUT ;
output C2TOP_OUT ;
output C2AND2_OUT ;
endmodule
```

External I/Os in a Module

It is recommended that you declare external I/Os in the top-level design. However, you can include external I/Os in a module without modifying the top-level code. This may be useful if you want to add a temporary external I/O in the module for simulation. To do this, explicitly instantiate IBUF/IBUFG/BUFGP and OBUF connections. Following are examples of code.

Note Do not directly connect these I/Os to module ports.

VHDL Example: Module Design with Inserted I/Os

```
library IEEE;
use IEEE.std_logic_1164.all;
entity module_a is port ( CLK_TOP : in std_logic;
    B2A_IN: in std_logic;
    TOP2A_IN: in std_logic;
    C2A_IN: in std_logic;
    MODA_DATA : in std_logic;
    MODA_CLK : in std_logic;
    MODA_OUT : out std_logic;
    A2B_OUT: out std_logic;
    A2TOP_OBUFT_I_OUT: out std_logic;
    A2C_OUT: out std_logic) ;
end module_a;
architecture modular of module_a is
-- add your signal declarations here
signal Q0_OUT, Q1_OUT, Q2_OUT, Q3_OUT : std_logic;
signal AND4_OUT: std_logic ;
signal OR4_OUT : std_logic;
begin
AND4_OUT <= Q0_OUT and Q1_OUT and Q2_OUT and Q3_OUT ;
OR4_OUT <= Q0_OUT or Q1_OUT or Q2_OUT or Q3_OUT ;
TOP_CLK: process(CLK_TOP)
begin
if (CLK_TOP’event and CLK_TOP = ’1’) then
  Q0_OUT <= MODA_DATA ;
  Q2_OUT <= TOP2A_IN ;
  MODA_OUT <= OR4_OUT ;
  A2B_OUT <= AND4_OUT ;
end if;
end process TOP_CLK;
```

```
CLK_MODA: process(MODA_CLK)
begin
if (MODA_CLK’event and MODA_CLK = ’1’) then
  Q1_OUT <= B2A_IN ;
  Q3_OUT <= C2A_IN ;
  A2TOP_OBUFT_I_OUT <= AND4_OUT ;
  A2C_OUT <= OR4_OUT ;
end if;
end process CLK_MODA;
end modular;
```

Verilog Example: Module Design with Inserted I/Os

In the following example, the module has two external inputs (IPAD_MODA_CLK and IPAD_MODA_DATA) and one external output (OPAD_MODA_OUT). These external I/Os, IBUF, OBUF, and BUFGP are instantiated.

The lower-level port declaration is different from the top-level declaration of module_a. Lower-level module_a has three additional ports. With Modular Design, NGDBuild ignores this port mismatch and uses module_a.edf to describe module_a. These I/Os will be present in the design and available for simulation.

```
module module_a ( CLK_TOP, B2A_IN, TOP2A_IN, C2A_IN, MODA_DATA,
MODA_CLK, MODA_OUT, A2B_OUT, A2TOP_OBUFT_I_OUT, A2C_OUT);
input CLK_TOP ;
input B2A_IN ;
input TOP2A_IN ;
input C2A_IN ;
input MODA_DATA, MODA_CLK;
output MODA_OUT;
output A2B_OUT ;
output A2TOP_OBUFT_I_OUT ;
output A2C_OUT ;
// add your declarations here
reg Q0_OUT, Q1_OUT, Q2_OUT, Q3_OUT ;
reg A2B_OUT, A2TOP_OBUFT_I_OUT, A2C_OUT ;
reg MODA_OUT;
wire AND4_OUT ;
wire OR4_OUT ;
// add your code here
assign AND4_OUT = Q0_OUT && Q1_OUT && Q2_OUT && Q3_OUT ;
assign OR4_OUT = Q0_OUT || Q1_OUT || Q2_OUT || Q3_OUT ;
always @ (posedge CLK_TOP)
begin : TOP_CLK
  Q0_OUT <= MODA_DATA ;
  Q2_OUT <= TOP2A_IN ;

  MODA_OUT <= OR4_OUT ;
  A2B_OUT <= AND4_OUT ;
end
always @ (posedge MODA_CLK)
begin : CLK_MODA
  Q1_OUT <= B2A_IN ;
  Q3_OUT <= C2A_IN ;

  A2TOP_OBUFT_I_OUT <= AND4_OUT ;
  A2C_OUT <= OR4_OUT ;

end
endmodule
```

Chapter 4

PARTGen

The PARTGen program is compatible with the following Xilinx devices.

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA
- CoolRunner™ XPLA3 / -II
- XC9500™ / XL / XV

This chapter describes PARTGen. The chapter contains the following sections.

- "PARTGen Overview"
- "PARTGen Syntax"
- "PARTGen Input Files"
- "PARTGen Output Files"
- "PARTGen Options"
- "Partlist.xct File"
- "PKG File"

PARTGen Overview

The PARTGen command displays various levels of information about installed Xilinx devices and families depending on which options are selected.

PARTGen Syntax

Following is the syntax for PARTGen:

```
partgen [options]
```

options can be any number of the options listed in the "PARTGen Options" section. They do not need to be listed in any particular order. Use spaces to separate multiple options.

PARTGen Input Files

PARTGen does not have any user input files.

PARTGen Output Files

PARTGen optionally outputs the following files if you use the –p and –v options:

- XCT file—This file contains detailed information about architectures and devices. See the "Partlist.xct File" section for a detailed description.
- PKG files—These files correlate IOBs with output pin names. The –p option generates a three column entry describing the pins. The –v option adds four more columns of descriptive pin information. See the "PKG File" section.

PARTGen Options

This section describes the command line options and how they affect the behavior of PARTGen.

–arch (Print Information for Specified architecture)

```
–arch architecture_name
```

The –arch option prints a list of devices, packages, and speeds for a specified architecture that has been installed.

Valid entries for *architecture_name* and the corresponding device product name are shown in the following table:

Table 4-1 Values for *architecture_name*

architecture_name	Corresponding Device Product Name
virtex	Virtex
virtex2	Virtex-II
virtex2p	Virtex-II PRO
virtexe	Virtex-E
spartan	Spartan
spartan2	Spartan-II
spartan2e	Spartan-IIE
spartanxl	SpartanXL
xc4000e	XC4000E
xc4000l	XC4000L
xc4000ex	XC4000EX
xc4000xl	XC4000XL
xc4000xla	XC4000XLA
xc9500	XC9500
xc9500xl	XC9500XL
xc9500xv	XC9500XV
xpla3	CoolRunner XPLA3
xbr	CoolRunner XPLA3-II

For example, entering the command partgen -arch spartan displays the following information:

```
s05             SPEEDS:         -4     -3
        PC84
        VQ100
s10             SPEEDS:         -4     -3
        PC84
        VQ100
        TQ144
s20             SPEEDS:         -4     -3
```

```
        VQ100
        TQ144
        PQ208
s30             SPEEDS:         -4    -3
        VQ100
        TQ144
        PQ208
        PQ240
        BG256
s40             SPEEDS:         -4    -3
        PQ208
        PQ240
        BG256
```

–i (Print a List of Devices, Packages, and Speeds)

The –i option prints out a list of devices, packages, and speeds that have been installed. Following is a portion of a sample display:

```
s05              SPEEDS:    -4    -3
      PC84
      VQ100
s10              SPEEDS:    -4    -3
      PC84
      VQ100
      TQ144
s20              SPEEDS:    -4    -3
      VQ100
      TQ144
      PQ208
s30              SPEEDS:    -4    -3
      VQ100
      TQ144
      PQ208
      PQ240
      BG256
.
.
.
```

–p (Creates Package file and Partlist.xct File)

–p *name*

The –p option generates a partlist.xct file for the specified name and also creates package files. All files are placed in the current working directory. Valid name entries include architectures, devices, and parts. Following are example command line entries of each type:

`–p virtex` (Architecture)

`–p xcv400` (Device)

`–p v400bg432` (Part)

If an architecture, device, or part is not specified with the –p option, detailed information for every installed device is submitted to the partlist.xct file.

The –p option generates more detailed information than the -arch options but less information that the –v option. The –p and –v options are mutually exclusive, that is, you can specify one or the other but not both.

Following is a portion of a partlist.xct file generated with the partgen -p virtex command:

```
partVIRTEX V50bg256 NA.die v50bg256.pkg \
        NCLBROWS=16 NCLBCOLS=24 STYLE=VIRTEX \
        IN_FF_PER_IOB=1 OUT_FF_PER_IOB=1 \
        NFRAMES=1450 NBITSPERFRAME=324
partVIRTEX V50cs144 NA.die v50cs144.pkg \
        NCLBROWS=16 NCLBCOLS=24 STYLE=VIRTEX \
        IN_FF_PER_IOB=1 OUT_FF_PER_IOB=1 \
        NFRAMES=1450 NBITSPERFRAME=324
partVIRTEX V50fg256 NA.die v50fg256.pkg \
        NCLBROWS=16 NCLBCOLS=24 STYLE=VIRTEX \
        IN_FF_PER_IOB=1 OUT_FF_PER_IOB=1 \
        NFRAMES=1450 NBITSPERFRAME=324
partVIRTEX V50pq240 NA.die v50pq240.pkg \
        NCLBROWS=16 NCLBCOLS=24 STYLE=VIRTEX \
        IN_FF_PER_IOB=1 OUT_FF_PER_IOB=1 \
        NFRAMES=1450 NBITSPERFRAME=324
.
```

```
.
.
```

For a description of the entries in the partlist.xct file, see the "Partlist.xct File" section.

–v (Creates Packages and Partlist.xct File)

–v name

The –v option generates a partlist.xct file for the specified name and also creates packages files. Valid name entries include architectures, devices, parts. Following are example command line entries of each type:

–v virtex (Architecture)

–v xcv400 (Device)

–v v400bg432 (Part)

If no architecture, device, or part is specified with the –v option, information for every installed device is submitted to the partlist.xct file.

The –v option generates more detailed information than the –p option. The –p and –v options are mutually exclusive, that is, you can specify one or the other but not both.

Following is a portion of a partlist.xct file generated with the partgen -v virtex command:

```
partVIRTEX V50bg256 NA.die v50bg256.pkg \
        NCLBROWS=16 NCLBCOLS=24 STYLE=VIRTEX \
        IN_FF_PER_IOB=1 OUT_FF_PER_IOB=1 \
        NPADS_PER_ROW=3 NPADS_PER_COL=2 \
        NFRAMES=1450 NBITSPERFRAME=324 \
        NIOBS=196 NBIOBS=180 \
        SLICES_PER_CLB=2 \
        NUM_BLK_RAMS=8\
        NUM_BLK_RAM_COLS=2 BLK_RAM_COL_0=0 BLK_RAM_COL_1=24 \
        BLK_RAM_SIZE=4096x1 BLK_RAM_SIZE=2048x2 BLK_RAM_SIZE=1024x4
BLK_RAM_SIZE=512x8 BLK_RAM_SIZE=256
```

```
x16 \
        FFS_PER_SLICE=2 \
        LATCHES_PER_SLICE=TRUE \
        NUM_LUTS_PER_SLICE=2 LUT_NAME=F LUT_SIZE=4 LUT_NAME=G LUT_SIZE=4 \
        MAX_LUT_PER_SLICE=5 \
        MAX_LUT_PER_CLB=19 \
        NUM_GLOBAL_BUFFERS=4 \
        GCLKBUF0=GCLKPAD0 GCLKBUF1=GCLKPAD1 GCLKBUF2=GCLKPAD2
GCLKBUF3=GCLKPAD3
\
        NUM_TBUFS_PER_ROW=52\
        NUM_CARRY_ELEMENTS_PER_SLICE=1\
        SPEEDGRADE=-6 LUTDELAY=573 IOB_IN_DELAY=658 IOB_OUT_DELAY=1887 \
        SPEEDGRADE=-5 LUTDELAY=641 IOB_IN_DELAY=737 IOB_OUT_DELAY=2065 \
        SPEEDGRADE=-4 LUTDELAY=738 IOB_IN_DELAY=847 IOB_OUT_DELAY=2300
                    .
                    .
                    .
```

For a description of the entries in the partlist.xct file, see the "Partlist.xct File" section.

Partlist.xct File

The partlist.xct file contains detailed information about architectures and devices.

The partlist.xct file is a series of part entries. There is one entry for every part supported in the installed software. The following subsections describe the information contained in the partlist.xct file.

Header

The first part is a header for the entry. The format of the entry looks like the following:

part *architecture family partname diename packagefilename*

Following is an example for the XCV50bg256:

```
partVIRTEX V50bg256 NA.die v50bg256.pkg
```

Device Attributes

The header is followed by a list of device attributes. Not all attributes are applicable to all devices.

- BSCAN pin mappings: TDK=PAD# TDI=PAD# TMS=PAD#
- CLB row and column sizes: NCLBROWS=# NCLBCOLS=#
- Sub-family designation: STYLE=sub_family
  (For example, STYLE = XC4000EX)
- Width of the edge decoder (found in the XC4000 family): EDGE_DECODER=#
- Input registers: IN_FF_PER_IOB=#
- Output registers: OUT_FF_PER_IOB=#
- Number of pads per row and per column: NPADS_PER_ROW=# NPADS_PER_COL=#
- Bitstream information:
  - Number of frames: NFRAMES=#
  - Number bits/frame: NBITSPERFRAME=#

The preceding bulleted items display for both the -p and -v options. The following bulleted items are displayed only when using the -v option:

- Number of IOBS in device: NIOBS=#
- Number of bonded IOBS: NBIOBS=#
- Slices per CLB: SLICES_PER_CLB=#

  For slice-based architectures, such as Virtex.

  (For non-slice based architectures, assume one slice per CLB)
- Flip-flops for each slice: FFS_PER_SLICE=#
- Latches for each slice: LATCHES_PER_SLICE={TRUE|FALSE}
- LUTs in a slice: LUT_NAME=name LUT_SIZE=#

- Number of global buffers: NUM_GLOBAL_BUFFERS=#

  (The number of places where a buffer can drive a global clock combination)

- External Clock IOB pins:

  - For the XC4000E family:

    ```
    BUFGP_TL=PAD#, BUFGP_BL=PAD#,
    BUFGP_BR=PAD#, BUFGP_TR=PAD#,
    BUFGS_TL=PAD#, BUFGS_BL=PAD#,
    BUFGS_BR=PAD#, BUFGS_TR=PAD#
    ```

  - For the XC4000EX / XC4000XL / XC4000XLA / SpartanXL families:

    ```
    BUFGLS_NNW=PAD#,
    BUFGLS_WNW=PAD#,
    BUFGLS_NNE=PAD#,
    BUFGLS_ENE=PAD#,
    BUFGLS_SSW=PAD#,
    BUFGLS_WSW=PAD#,
    BUFGLS_SSE=PAD#,
    BUFGLS_ESE=PAD#
    ```

  - For the Virtex, Virtex-E, and Spartan-II:

    ```
    GCLKBUF0=PAD#, GCLKBUF1=PAD#,
    GCLKBUF2=PAD#, GCLKBUF3=PAD#
    ```

  - For Virtex-II and Virtex-II PRO:

    ```
    BUFGMUX0P=PAD#, BUFGMUX1P=PAD#,
    BUFGMUX2P=PAD#, BUFGMUX3P=PAD#,
    BUFGMUX4P=PAD#, BUFGMUX5P=PAD#,
    BUFGMUX6P=PAD#, BUFGMUX7P=PAD#
    ```

- Block RAM:

  ```
  NUM_BLK_RAMS=#
  BLK_RAM_COLS=# BLK_RAM_COL0=# BLK_RAMCOL1=#
  BLK_RAM_COL2=# BLK_RAM_COL_3=#
  BLK_RAM_SIZE=4096x1 BLK_RAM_SIZE=2048x2
  BLK_RAM_SIZE=512x8 BLK_RAM_SIZE=256x16
  ```

Block RAM locations are given with reference to CLB columns. In the following example, Block RAM 5 is positioned in CLB column 32.

```
NUM_BLK_RAMS=10 BLK_RAM_COL_5=32
BLK_RAM_SIZE=4096X1
```

- Select RAM:

  ```
  NUM_SEL_RAMS=#    SEL_RAM_SIZE=#X#
  ```

- Select Dual Port RAM:

  ```
  SEL_DP_RAM={TRUE|FALSE}
  ```

  This field indicates whether the select RAM can be used as a dual port ram. The assumption is that the number of addressable elements is reduced by half, that is, the size of the select RAM in Dual Port Mode is half that indicated by SEL_RAM_SIZE.

- Speed grade information: SPEEDGRADE=#
- Typical delay across a LUT for each speed grade: LUTDELAY=#
- Typical IOB input delay: IOB_IN_DELAY=#
- Typical IOB output delay: IOB_OUT_DELAY=#
- Maximum LUT constructed in a slice:

  ```
  MAX_LUT_PER_SLICE=#
  ```

  (From all the LUTs in the slice)

- Max LUT constructed in a CLB: MAX_LUT_PER_CLB=#

  This field describes how wide a LUT can be constructed in the CLB from the available LUTs in the slice.

- Number of internal 3-state buffers in a device: NUM_TBUFFS=#

PKG File

The PKG files correlate IOBs with output pin names. The –p option generates a three column entry describing the pins. The –v option adds four more columns of descriptive pin information.

For example, the command partgen –p xc2v40 generates the package files: 2v40cs144.pkg and 2v40fg256.pkg. Following is a portion of the package file for the 2v40cs144:

```
package 2v40cs144
pin PAD96 D3
pin PAD2 A3
pin PAD3 C4
pin PAD4 B4
.
.
.
```

The first column contains either *pin* (user accessible pin) or *pkpin* (dedicated pin). The second column specifies the pin name. For user accessible pins, the name of the pin is the bonded pad name associated with an IOB on the device, or the name of a multi-purpose pin. For dedicated pins, the name is either the functional name of the pin, or N.C. indicating No Connection. The third column specifies the package pin.

The command partgen –v generates package (.pkg) files and generates a seven column entry describing the pins. The first three columns are described above.

The fourth column, IO_BANK, is a positive integer associated with a bank, or -1 for no bank association. The fifth column, specifying function name, consists of a string indicating how the pin is used. If the pin is dedicated, then the string will indicate a specific function. If the pin is a generic user pin, the string will be *IO*. If the pin is multipurpose, an underscore separated set of characters will make up the string. The sixth column indicates the closest CLB row or column to the pin, and appears in the form R[0-9]C[0-9]. The seventh column is composed of a string for each pin associated with a LVDS IOB. The string consists of and index and the letter M or S. Index values will go from 0 to the number of LVDS pairs. The value for a non-LVDS pin will default to N.A. The following are examples of the verbose pin descriptors in PARTGen:

```
pkpin N.C. D9 -1 N.C.N.A. N.A.
pkpin DONE M12 -1 DONE N.A.N.A.
pkpin VCCO N1 -1 VCCO N.A.N.A.
pin PAD2 B3 0 IO R0C0 0S
pin PAD17 AB23 7 IO_TDO R6C48 N.A.
```

Chapter 5

NGDBuild

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes the NGDBuild program. The chapter contains the following sections:

- "NGDBuild Overview"
- "NGDBuild Syntax"
- "NGDBuild Input Files"
- "NGDBuild Output Files"
- "NGDBuild Intermediate Files"
- "NGDBuild Options"

NGDBuild Overview

NGDBuild performs all the steps necessary to read a netlist file in XNF, EDIF, or NGC format and create an NGD file describing the logical design (a logical design is in terms of logic elements such as AND gates, OR gates, decoders, flip-flops, and RAMs). The NGD file resulting from an NGDBuild run contains both a logical description of the design reduced to Xilinx Native Generic Database (NGD) primitives and a description in terms of the original hierarchy expressed in the input netlist. The output NGD file can be mapped to the desired device family.

The following figure shows a simplified version of the NGDBuild design flow. NGDBuild invokes other programs that are not shown in the drawing. For a complete description of how NGDBuild works, see the "EDIF2NGD, XNF2NGD, and NGDBuild" appendix.

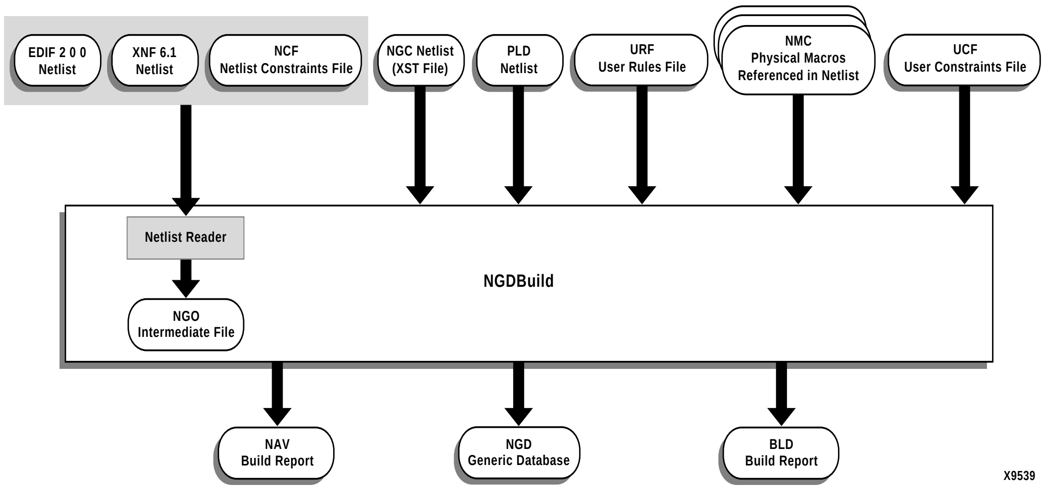


Figure 5-1 NGDBuild Design Flow

Converting a Netlist to an NGD File

NGDBuild performs the following steps to convert a netlist to an NGD file:

1. Reads the source netlist

   NGDBuild invokes the Netlister Launcher. The Netlist Launcher determines the type of the input netlist and starts the appropriate netlist reader program. The netlist readers incorporate NCF files associated with each netlist. NCF files contain timing and layout constraints for each module. The Netlister Launcher is described in detail in the "Netlister Launcher" section of the "EDIF2NGD, XNF2NGD, and NGDBuild" appendix.

2. Reduces all components in the design to NGD primitives

   NGDBuild merges components that reference other files. NGDBuild also finds the appropriate system library components, physical macros (NMC files), and behavioral models.

3. Checks the design by running a Logical Design Rule Check (DRC) on the converted design

   The Logical DRC is a series of tests on the logical design. It is described in the "Logical Design Rule Check" chapter.

4. Writes an NGD file as output

Note This procedure, the Netlister Launcher, and the netlist reader programs are described in more detail in the "EDIF2NGD, XNF2NGD, and NGDBuild" appendix.

NGDBuild Syntax

The following command reads the design into the Xilinx Development system and converts it to an NGD file:

```
ngdbuild [options] design_name [ngd_file[.ngd]]
```

options can be any number of the NGDBuild options listed in the "NGDBuild Options" section. They can be listed in any order. Separate multiple options with spaces.

design_name is the top-level name of the design file you want to process. To ensure the design processes correctly, specify a file extension for the input file. Use one of the legal file extensions specified in the "NGDBuild Input Files" section. Using an incorrect or nonexistent file extension causes NGDBuild to fail without creating an NGD file. If you use an incorrect file extension, NGDBuild may issue an "unexpanded" error.

Note If you are using an NGC file as your input design, it is recommended that you specify the .ngc extension. If NGDBuild finds an EDIF netlist or NGO file in the project directory, it does not check for an NGC file.

ngd_file[**.ngd**] is the output file in NGD format. The output file name, its extension, and its location are determined as follows:

- If you do not specify an output file name, the output file has the same name as the input file, with an .ngd extension.
- If you specify an output file name with no extension, NGDBuild appends the .ngd extension to the file name.
- If you specify a file name with an extension other than .ngd, you get an error message and NGDBuild does not run.
- If the output file already exists, it is overwritten with the new file.

NGDBuild Input Files

NGDBuild uses the following files as input:

- Design file—The input design can be an XNF, EDIF 2 0 0, PLD, or NGC netlist file. If the input netlist is in another format that the Netlister Launcher recognizes, the Netlister Launcher invokes the program necessary to convert the netlist to EDIF or XNF format, then invokes the appropriate netlist reader, EDIF2NGD or XNF2NGD.

With the default Netlister Launcher options, NGDBuild recognizes and processes files with the extensions shown in the following table. NGDBuild searches the top-level design netlist directory for a netlist file with one of these extensions in the order shown in this table. For example, NGDBuild searches for an XTF file first. If one is not found, it then searches for an XG file.

File Type	Recognized Extensions
XNF	.xtf, .xg, .xff, .sxnf, .xnf
EDIF	.sedif, .edn, .edf, .edif
NGC	.ngc
PLD	.pld

Note Remove all out of date netlist files from your directory. Obsolete netlist files may cause errors in NGDBuild.

- UCF file—The User Constraints File is an ASCII file that you create. You can create this file by hand or by using the Constraints Editor. See the online Help provided with the Constraints Editor for more information. The UCF file contains timing and layout constraints that affect how the logical design is implemented in the target device. The constraints in the file are added to the information in the output NGD file. For detailed information on constraints, see the *Constraints Guide*.

  By default, NGDBuild reads the constraints in the UCF file automatically if the UCF file has the same base name as the input design file and a .ucf extension. You can override the default behavior and specify a different constraints file by entering a –uc option to the NGDBuild command line. See the "–uc (User Constraints File)" section for more information.

  Note NGDBuild only allows one UCF file as input.

- NCF file—The Netlist Constraints File is produced by a CAE vendor toolset. This file contains constraints specified within the toolset. The netlist reader invoked by NGDBuild reads the constraints in this file if the NCF file has the same name as the input EDIF or XNF netlist file. It adds the constraints to the intermediate NGO file and the output NGD file. NGC netlist files do not have corresponding NCF files. NCF information is contained within the NGC file itself.

Note If the NGO file for a netlist file is up to date, NGDBuild looks for an NCF file with the same base name as the netlist in the netlist directory and compares the timestamp of the NCF file against that of the NGO file. If the NCF file is newer, XNF2NGD or EDIF2NGD is run again. However, if an NCF file existed on a previous run of NGDBuild and the NCF file was deleted, NGDBuild does not detect that XNF2NGD or EDIF2NGD must be run again. In this case, you must use the –nt on option to force a rebuild. See the "–nt (Netlist Translation Type)" section for more information.

- URF file — The User Rules File is an ASCII file that you create. The Netlister Launcher reads this file to determine the acceptable netlist input files, the netlist readers that read these files, and the default netlist reader options. This file also allows you to specify third-party tool commands for processing designs. The user rules file can add to or override the rules in the system rules file.

  You can specify the location of the user rules file with the NGDBuild –ur option. The user rules file must have a .urf extension. See the "–ur (Read User Rules File)" section for more information. The user rules file is described in the "User Rules File" section of the "EDIF2NGD, XNF2NGD, and NGDBuild" appendix.

- NGC file—This binary file can be used as a top-level design file or as a module file:
  - ♦ Top-level design file

    This file is output by the Xilinx Synthesis Technology (XST) tool. See the description of design files earlier in this section for details.

    Note This is not a "true" netlist file. However, it is referred to as a netlist in this context to differentiate it from the NGC module file.

  - ♦ Module file

    In HDL design flows, LogiBLOX creates an NGC file to define each module. This file contains the implementation of a module in the design. If an NGC file exists for a module, NGDBuild reads this file directly, without looking for a source EDIF or XNF netlist.

- NMC files—These physical macros are binary files that contain the implementation of a physical macro instantiated in the design. NGDBuild reads the NMC file to create a functional simulation model for the macro.
- MEM files—These LogiBLOX memory definition files are text files that define the contents of LogiBLOX memory modules. NGDBuild reads MEM files in design flows where LogiBLOX does not create NGC files directly. See the "Module Descriptions" chapter of the *LogiBLOX Guide* for details.

Unless a full path is provided to NGDBuild, it searches for netlist, NGC, NMC, and MEM files in the following locations:

- The working directory from which NGDBuild was invoked
- The path specified for the top-level design netlist on the NGDBuild command line
- Any path specified with the –sd switch on the NGDBuild command line

NGDBuild Output Files

NGDBuild creates the following files as output:

- NGD file—This binary file contains a logical description of the design in terms of both its original components and hierarchy and the NGD primitives to which the design reduces.
- BLD file—This build report file contains information about the NGDBuild run and about the subprocesses run by NGDBuild. Subprocesses include EDIF2NGD, XNF2NGD, and programs specified in the URF file. The BLD file has the same root name as the output NGD file and a .bld extension. The file is written into the same directory as the output NGD file.
- NAV file—This build report file contains the same information as the BLD file. However, you use the Error Viewer GUI to read this report, and you can click a name to navigate back to the instance or net in the source design as seen in the third party tool.

  Enter the following on the command line to launch the Error Viewer GUI. See the online Help provided with the GUI for more information.

  ```
  errview design_name_ngdbuild.nav
  ```

The NAV file has the same root name as the output NGD file and a _ngdbuild.nav extension. The file is written into the same directory as the output NGD file.

Note If you attach a pull-up or pull-down property on a pad net in your UCF file, a comment in the BLD and NAV files indicates that NGDBuild added a pull-up or pull-down instance to the net.

NGDBuild Intermediate Files

NGO files—These binary files contain a logical description of the design in terms of its original components and hierarchy. These files are created when NGDBuild reads the input EDIF or XNF netlist. If these files already exist, NGDBuild reads the existing files. If these files do not exist or are out of date, NGDBuild creates them.

NGDBuild Options

This section describes the NGDBuild command line options.

–a (Add PADs to Top-Level Port Signals)

If the top-level input netlist is in EDIF format, the –a option causes NGDBuild to add a PAD symbol to every signal that is connected to a port on the root-level cell. This option has no effect on lower-level netlists or on a top-level XNF netlist.

Using the –a option depends on the behavior of your third-party EDIF writer. If your EDIF writer treats pads as instances (like other library components), do not use –a. If your EDIF writer treats pads as hierarchical ports, use –a to infer actual pad symbols. If you do not use –a where necessary, logic may be improperly removed during mapping.

For EDIF files produced by Mentor Graphics and Cadence, the –a option is set automatically; you do *not* have to enter –a explicitly for these vendors.

Note The NGDBuild –a option corresponds to the EDIF2NGD –a option. If you run EDIF2NGD on the top-level EDIF netlist separately, rather than allowing NGDBuild to run EDIF2NGD, you must use the two –a options consistently. If you previously ran NGDBuild on your design and NGO files are present, you must use the –nt on option the first time you use –a. This forces a rebuild of the NGO files, allowing EDIF2NGD to run the –a option.

–aul (Allow Unmatched LOCs)

By default (without the –aul option), NGDBuild generates an error if the constraints specified for pin, net, or instance names in the UCF or NCF file cannot be found in the design. If this error occurs, an NGD file is not written. If you enter the –aul option, NGDBuild generates a warning instead of an error for LOC constraints and writes an NGD file.

You may want to run NGDBuild with the –aul option if your constraints file includes location constraints for pin, net, or instance names that have not yet been defined in the HDL or schematic. This allows you to maintain one version of your constraints files for both partially complete and final designs.

Note When using this option, make sure you do not have misspelled net or instance names in your design. Misspelled names may cause inaccurate placing and routing.

–dd (Destination Directory)

```
-dd ngo_directory
```

The –dd option specifies the directory for intermediate files (design NGO files and netlist files). If the –dd option is not specified, files are placed in the current directory.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–i (Ignore UCF File)

By default (without the –i option), NGDBuild reads the constraints in the UCF file automatically if the UCF file in the top-level design netlist directory has the same base name as the input design file and a .ucf extension. The –i option ignores the UCF file.

Note If you use this option, do not use the –uc option.

–l (Libraries to Search)

```
–l libname
```

The –l option indicates the list of libraries to search when determining what library components were used to build the design. This option is passed to the appropriate netlist reader. The information allows NGDBuild to determine the source of the design's components so it can resolve the components to NGD primitives.

You can specify multiple libraries by entering multiple **–l** *libname* entries on the NGDBuild command line.

The allowable entries for *libname* are the following:

- **xilinxun** (Xilinx Unified library)
- **synopsys**

Note You do not have to enter xilinxun with a –l option. The Xilinx Development System tools automatically access these libraries. In cases where NGDBuild automatically detects Synopsys designs (for example, the netlist extension is .sxnf or .sedif), you do not have to enter synopsys with a –l option.

–modular assemble (Module Assembly)

```
–modular assemble -pimpath pim_directory_path
-use_pim module_name1 -use_pim module_name2 ...
```

Note This option is supported for Virtex/-II/-II PRO/-E and Spartan-II/-IIE device families only.

The –modular assemble option starts the final phase of the Modular Design flow. In this "Final Assembly" phase, the team leader uses this option to create a fully expanded NGD file that contains logic from the top-level design and each of the Physically Implemented Modules (PIMs). The team leader then implements this NGD file.

Run this option from the top-level design directory.

If you are running the standard Modular Design flow, you do not need to use the –pimpath option. If you do not use the –usepim option, NGDBuild searches the PIM directory's subdirectories for NGO files with names that match their subdirectory. It assembles the final design using these NGO files.

If you are running Modular Design in a Partial Assembly flow, use the –pimpath option to specify the directory that contains the PIMs. Use the –usepim option to identify all the modules in the PIM directory that have been published. Be sure to use exact names of the PIMs, including the proper spelling and capitalization. The input design file should be the NGO file of the top-level design.

Note When running Modular Design in a Partial Assembly flow, you must use the –modular assemble option with the –u option. For details, see the "Running the Partial Design Assembly Flow" section of the "Modular Design" chapter.

See the "Assembling the Modules" section of the "Modular Design" chapter for more information.

–modular initial (Initial Budgeting of Modular Design)

Note This option is supported for Virtex / -II / -II PRO / -E and Spartan-II / -IIE device families only.

The –modular initial option starts the first phase of the Modular Design flow. In this "Initial Budgeting" phase, the team leader uses this option to generate an NGO and NGD file for the top-level design with all of the instantiated modules represented as unexpanded blocks. After running this option, the team leader sets up initial budgeting for the design. This includes assigning top-level timing constraints and location constraints for various resources, including each module, using the Floorplanner and Constraints Editor tools.

Note You cannot use the NGD file for mapping.

Run this option from the top-level design directory. The input design file should be an EDIF netlist or an NGC netlist from XST.

See the "Running Initial Budgeting" section of the "Modular Design" chapter for more information.

–modular module (Active Module Implementation)

```
–modular module -active module_name
```

Note This option is supported for Virtex/-II/-II PRO/-E and Spartan-II/-IIE device families only. You *cannot* use NCD files from previous software releases with Modular Design in this release. You must generate new NCD files with the current release of the software.

The –modular module option starts the second phase of the Modular Design flow. In this "Active Module Implementation" phase, each team member creates an NGD file with just the specified "active" module expanded. This NGD file is named after the top-level design.

Run this option from the active module directory. This directory should include the active module netlist file and the top-level UCF file generated during the Initial Budgeting phase. You must specify the name of the active module after the –active option, and use the top-level NGO file as the input design file.

After running this option, you can then run MAP and PAR to create a Physically Implemented Module (PIM). Then, you must run PIMCreate to publish the PIM to the PIMs directory. PIMCreate copies the local, implemented module file, including the NGO, NGM and NCD files, to the appropriate module directory inside the PIMs directory and renames the files to *module_name.extension*. To run PIMCreate, type the following on the command line:

```
pimcreate pim_directory -ncd
design_name_routed.ncd
```

See the "Implementing an Active Module" section of the "Modular Design" chapter for more information.

Note When running Modular Design in an Incremental Guide flow, run NGDBuild with the –pimpath and –use_pim options normally reserved for the –modular assemble option. See the "Running the Incremental Guide Flow" section of the "Modular Design" chapter for more information.

–nt (Netlist Translation Type)

```
–nt {timestamp | on | off}
```

The –nt option determines how timestamps are treated by the Netlister Launcher when it is invoked by NGDBuild. A timestamp is information in a file that indicates the date and time the file was created. The timestamp option (which is the default if no –nt option is specified) instructs the Netlister Launcher to perform the normal timestamp check and update NGO files according to their timestamps. The on option translates netlists regardless of timestamps (rebuilding all NGO files), and the off option does not rebuild an existing NGO file, regardless of its timestamp.

–p (Part Number)

```
–p part
```

The –p option specifies the part into which the design is implemented.The –p option can specify an architecture only, a complete part specification (device, package, and speed), or a partial specification (for example, device and package only).

The syntax for the –p option is described in the "–p (Part Number)" section of the "Introduction" chapter. Examples of part entries are **XCV50-TQ144** and **XCV50-TQ144-5**.

When you specify the *part*, the NGD file produced by NGDBuild is optimized for mapping into that architecture.

You do not have to specify a –p option if your NGO file already contains information about the desired vendor and family (for example, if you placed a PART property in a schematic or a CONFIG PART statement in a UCF file). However, you can override the information in the NGO file with the –p option when you run NGDBuild.

Note If you are running the Modular Design flow and are targeting a part different from the one specified in your source design, you must specify the part type using the -p option *every time* you run NGDBuild.

–quiet (Report Warnings and Errors Only)

The –quiet option reduces NGDBuild screen output to warnings and errors only. This option is useful if you only want a summary of the NGDBuild run.

–r (Ignore LOC Constraints)

The –r option eliminates all location constraints (LOC=) found in the input netlist or UCF file. Use this option when you migrate to a different device or architecture, because locations in one architecture may not match locations in another.

Note If you previously ran NGDBuild on your design and NGO files are present, you must use the –nt on option the first time you use –r. This forces a rebuild of the NGO files, allowing NGDBuild to run EDIF2NGD or XNF2NGD to remove location constraints.

–sd (Search Specified Directory)

```
–sd search_path
```

The –sd option adds the specified *search_path* to the list of directories to search when resolving file references (that is, files specified in the schematic with a FILE=*filename* property) and when searching for netlist, NGO, NGC, NMC, and MEM files. You do not have to specify a search path for the top-level design netlist directory, because it is automatically searched by NGDBuild.

The *search_path* must be separated from the –sd by spaces or tabs (for example, **–sd designs** is correct, **–sddesigns** is not). You can specify multiple –sd options on the command line. Each must be preceded with –sd; you cannot combine multiple search_path specifiers after one –sd. For example, the following syntax is *not* acceptable.

```
–sd /home/macros/counter /home/designs/pal2
```

The following syntax is acceptable.

```
–sd /home/macros/counter –sd /home/designs/pal2
```

–u (Allow Unexpanded Blocks)

In the default behavior of NGDBuild (without the –u option), NGDBuild generates an error if a block in the design cannot be expanded to NGD primitives. If this error occurs, an NGD file is not written. If you enter the –u option, NGDBuild generates a warning instead of an error if a block cannot be expanded, and writes an NGD file containing the unexpanded block.

You may want to run NGDBuild with the –u option to perform preliminary mapping, placement and routing, timing analysis, or simulation on the design even though the design is not complete. To ensure the unexpanded blocks remains in the design when it is mapped, run the MAP program with the –u (Do Not Remove Unused Logic) option, as described in the "–u (Do Not Remove Unused Logic)" section of the "MAP" chapter.

–uc (User Constraints File)

```
-uc ucf_file[.ucf]
```

The –uc option specifies a User Constraints File (UCF) for the Netlister Launcher to read. The UCF file contains timing and layout constraints that affect the way the logical design is implemented in the target device.

The user constraints file must have a .ucf extension. If you specify a user constraints file without an extension, NGDBuild appends the .ucf extension to the file name. If you specify a file name with an extension other than .ucf, you get an error message and NGDBuild does not run.

If you do not enter a –uc option and a UCF file exists with the same base name as the input design file and a .ucf extension, NGDBuild automatically reads the constraints in this UCF file.

See the *Constraints Guide* for more information on the UCF file.

Note NGDBuild only allows one UCF file as input. Therefore, you cannot specify multiple –uc options on the command line. Also, if you use this option, do not use the –i option.

–ur (Read User Rules File)

```
-ur rules_file[.urf]
```

The –ur option specifies a user rules file for the Netlister Launcher to access. This file determines the acceptable netlist input files, the netlist readers that read these files, and the default netlist reader options. This file also allows you to specify third-party tool commands for processing designs.

The user rules file must have a .urf extension. If you specify a user rules file with no extension, NGDBuild appends the .urf extension to the file name. If you specify a file name with an extension other than .urf, you get an error message and NGDBuild does not run.

The user rules file is described in the "User Rules File" section of the "EDIF2NGD, XNF2NGD, and NGDBuild" appendix.

–verbose (Report All Messages)

The –verbose option enhances NGDBuild screen output to include all messages output by the tools run: NGDBuild, the netlist launcher, and the netlist reader. This option is useful if you want to review details about the tools run.

Logical Design Rule Check

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes the Logical Design Rule Check (DRC). The chapter contains the following sections:

- "Logical DRC Overview"
- "Logical DRC Checks"

Logical DRC Overview

The Logical Design Rule Check (DRC), also known as the NGD DRC, comprises a series of tests to verify the logical design in the Native Generic Database (NGD) file. The Logical DRC performs device-independent checks.

The Logical DRC generates messages to show the status of the tests performed. Messages can be error messages (for conditions where the logic will not operate correctly) or warnings (for conditions where the logic is incomplete).

The Logical DRC runs automatically at the following times:

- At the end of NGDBuild, before NGDBuild writes out the NGD file

  NGDBuild writes out the NGD file if DRC warnings are discovered, but does not write out an NGD file if DRC errors are discovered.

- At the end of NGD2EDIF, NGD2VER, or NGD2VHDL, before writing out the netlist file

  The netlist writers do not perform the entire DRC. They only perform the Net and Name checks. A netlist writer writes out a netlist file even if DRC warnings or errors are discovered.

Logical DRC Checks

The Logical DRC performs the following types of checks:

- Block check
- Net check
- Pad check
- Clock buffer check
- Name check
- Primitive pin check

The following sections describe these tests.

Block Check

The block check verifies that each terminal symbol in the NGD hierarchy (that is, each symbol that is not resolved to any lower-level components) is an NGD primitive. A block check failure is treated as an error. As part of the block check, the DRC also checks user-defined properties on symbols and the values on the properties to make sure they are legal.

Net Check

The net check determines the number of NGD primitive output pins (drivers), 3-state pins (drivers), and input pins (loads) on each signal in the design. If a signal does not have at least one driver (or one

3-state driver) and at least one load, a warning is generated. An error is generated if a signal has multiple non-3-state drivers or any combination of 3-state and non-3-state drivers. As part of the net check, the DRC also checks user-defined properties on signals and the values on the properties to make sure they are legal.

Pad Check

The pad check verifies that each signal connected to pad primitives obeys the following rules.

- If the PAD is an input pad, the signal to which it is connected can only be connected to the following types of primitives:
    - Buffers
    - Clock buffers
    - PULLUP
    - PULLDOWN
    - KEEPER
    - BSCAN

  The input signal can be attached to multiple primitives, but only one of each of the above types. For example, the signal can be connected to a buffer primitive, a clock buffer primitive, and a PULLUP primitive, but it cannot be connected to a buffer primitive and two clock buffer primitives. Also, the signal cannot be connected to both a PULLUP primitive and a PULLDOWN primitive. Any violation of the rules above results in an error, with the exception of signals attached to multiple pull-ups or pull-downs, which produces a warning. A signal which is not attached to any of the above types of primitives also produces a warning.
- If the PAD is an output pad, the signal it is attached to can only be connected to one of the following primitive outputs:
    - A single buffer primitive output
    - A single 3-state primitive output
    - A single BSCAN primitive

In addition, the signal can also be connected to one of the following primitives:

- ♦ A single PULLUP primitive
- ♦ A single PULLDOWN primitive
- ♦ A single KEEPER primitive

Any other primitive output connections on the signal results in an error.

If the condition above is met, the output PAD signal may also be connected to one clock buffer primitive *input*, one buffer primitive *input*, or both.

- If the PAD is a bidirectional or unbonded pad, the signal it is attached to must obey the rules stated above for input and output pads. Any other primitive connections on the signal results in an error. The signal connected to the pad must be configured as both an input and an output signal; if it is not, you receive a warning.
- If the signal attached to the pad has a connection to a top-level symbol of the design, that top-level symbol pin must have the same type as the pad pin, except that output pads can be associated with 3-state top-level pins. A violation of this rule results in a warning.
- If a signal is connected to multiple pads, an error is generated. If a signal is connected to multiple top-level pins, a warning is generated.

Clock Buffer Check

The clock buffer configuration check verifies that the output of each clock buffer primitive is connected to only inverter, flip-flop or latch primitive clock inputs, or other clock buffer inputs. Violations are treated as warnings.

Name Check

The name check verifies the uniqueness of names on NGD objects using the following criteria:

- Pin names must be unique within a symbol. A violation results in an error.
- Instance names must be unique within the instance's position in the hierarchy (that is, a symbol cannot have two symbols with the same name under it). A violation results in a warning.
- Signal names must be unique within the signal's hierarchical level (that is, if you push down into a symbol, you cannot have two signals with the same name). A violation results in a warning.
- Global signal names must be unique within the design. A violation results in a warning.

Primitive Pin Check

The primitive pin check verifies that certain pins on certain primitives are connected to signals in the design. The following table shows which pins are tested on each NGD primitive type.

Table 6-1 Checked Primitive Pins

NGD Primitive	Pins Checked
X_MUX	SEL
X_TRI	IN, OUT, and CTL
X_FF	IN, OUT, and CLK
X_LATCH	IN, OUT, and CLK
X_IPAD	PAD
X_OPAD	PAD
X_BPAD	PAD

Note If one of these pins is not connected to a signal, you receive a warning.

Chapter 7

MAP

This program is compatible with the following families:

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA

This chapter describes MAP. The chapter contains the following sections:

MAP Overview

The MAP program maps a logical design to a Xilinx FPGA. The input to MAP is an NGD file, which contains a logical description of the design in terms of both the hierarchical components used to develop the design and the lower level Xilinx primitives. The NGD file also contains any number of NMC (macro library) files, each of which contains the definition of a physical macro. MAP first performs a logical DRC (Design Rule Check) on the design in the NGD file. MAP then maps the logic to the components (logic cells, I/O cells, and other components) in the target Xilinx FPGA. The output design is an NCD (Native Circuit Description) file—a physical representation of the design mapped to the components in the Xilinx FPGA. The NCD file can then be placed and routed.

The following figure shows the MAP design flow:

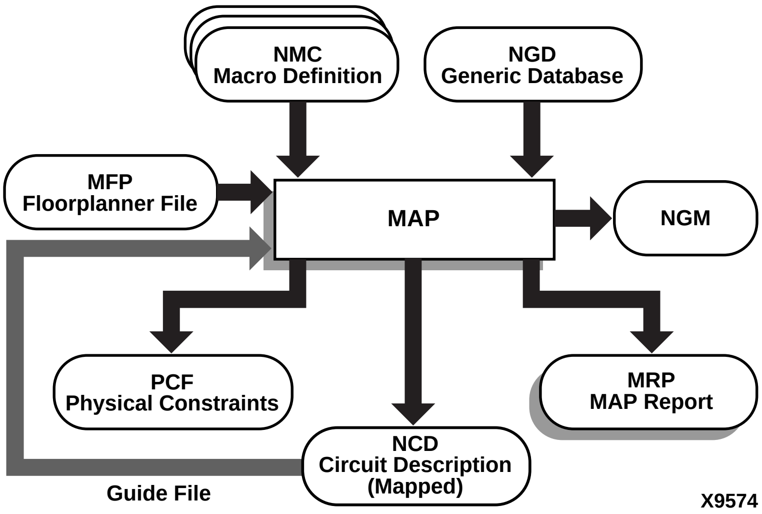


Figure 7-1 MAP

MAP Syntax

The following syntax maps your design:

```
map [options] infile[.ngd] [pcf_file[.pcf]]
```

Options can be any number of the MAP options listed in the "MAP Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

infile[**.ngd**] is the input NGD file. You do not have to enter the .ngd extension.

pcf_file[**.pcf**] is the output Physical Constraints File in PCF format. A constraints file name is optional on the command line, but if one is entered it must be entered after the input file name. You do not have to enter the .pcf extension. The constraints file name and its location are determined in the following ways:

- If you do not specify a physical constraints file name on the command line, the physical constraints file has the same name as the output file, with a .pcf extension. The file is placed in the output file's directory.
- If you specify a physical constraints file with no path specifier (for example, **cpu_1.pcf** instead of **/home/designs/cpu_1.pcf**), the .pcf file is placed in the current working directory.
- If you specify a physical constraints file name with a full path specifier (for example, **/home/designs/cpu_1.pcf**), the physical constraints file is placed in the specified directory.
- If the physical constraints file already exists, MAP reads the file, checks it for syntax errors, and overwrites the schematic-generated section of the file. MAP also checks the user-generated section for errors and corrects errors by commenting out physical constraints in the file or by halting the operation. If no errors are found in the user-generated section, the section remains the same.

For a discussion of the output file name and its location, see the "–o (Output File Name)" section.

MAP Input Files

MAP uses the following files as inputs:

- NGD file—Native Generic Database file. This file contains a logical description of the design expressed both in terms of the hierarchy used when the design was first created and in terms of lower-level Xilinx primitives to which the hierarchy resolves. The file also contains all of the constraints applied to the design during design entry or entered in a UCF (User Constraints File). The NGD file is created by NGDBuild.
- NMC files—Macro library files. An NMC file contains the definition of a physical macro. When there are macro instances in the NGD design file, NMC files are used to define the macro instances. There is one NMC file for each *type* of macro in the design file.

- Guide NCD file—An optional input file generated from a previous MAP run. An NCD file contains a physical description of the design in terms of the components in the target Xilinx device. A guide NCD file is an output NCD file from a previous MAP run that is used as an input to guide a later MAP run.
- Guide NGM file—A binary design file containing all of the data in the input NGD file as well as information on the physical design produced by the mapping. See "Guided Mapping" section for details.
- MFP—Map Floorplanner File, which is generated by the Floorplanner, specified as an input file with the –fp option. The MFP file is used as a guide file for mapping. To create a Map Floorplanner File, you must first have generated an NGD file and a mapped NCD file. When you have run MAP to generate an NCD file, you can open the mapped NCD file in the Floorplanner, modify the placement of components, and then generate an MFP file. You can then use the MFP file as an input file with the –fp map option.
- MDF file—MAP Directive File. The MDF is an optional input file used for guided mapping. The MDF file describes how logic was decomposed when the guide design was mapped. MAP uses the hints in the MDF as a guide for logic decomposition in the guided mapping run.

MAP Output Files

Output from MAP consists of the following files:

- NCD file—Native Circuit Description. A physical description of the design in terms of the components in the target Xilinx device. For a discussion of the output NCD file name and its location, see the "–o (Output File Name)" section.
- PCF (Physical Constraints) file—an ASCII text file containing the constraints specified during design entry expressed in terms of physical elements. The physical constraints in the PCF file are expressed in Xilinx's constraint language.

  MAP either creates a PCF file if none exists or rewrites an existing file by overwriting the schematic-generated section of the file (between the statements SCHEMATIC START and SCHEMATIC END). For an existing physical constraints file, MAP also checks the user-generated section for syntax errors, and signals errors by halting the operation. If no errors are found in the user-generated section, the section is unchanged.

- NGM file—a binary design file containing all of the data in the input NGD file as well as information on the physical design produced by the mapping. The NGM file is used to correlate the back-annotated design netlist to the structure and naming of the source design.
- MRP (MAP report) file—a file containing information about the MAP command run. The MRP file lists any errors and warnings found in the design, lists design attributes specified, details on how the design was mapped (for example, the logic that was removed or added and how signals and symbols in the logical design were mapped into signals and components in the physical design). The file also supplies statistics about component usage in the mapped design. See "MAP Report (MRP) File" section for more details.
- MDF (MAP Directive File)—a file describing how logic was decomposed when the design was mapped. In guided mapping, MAP uses the hints in the MDF as a guide for logic decomposition.

The MRP, MDF and NGM files produced by a MAP run all have the same name as the output file, with the appropriate extension. If the MRP, MDF or NGM files already exist, they are overwritten by the new files.

MAP Options

The following table shows which architectures can be used with each option.

Table 7-1 Map Options and Architectures

Options	Architectures
–b	Spartan, xc4000e/l
–bp	Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E
–c	All FPGA architectures
–cm	All FPGA architectures

Table 7-1 Map Options and Architectures

Options	Architectures
–detail	All FPGA architectures
–f	All FPGA architectures
–fp	All FPGA architectures
–gf	All FPGA architectures
–gm	All FPGA architectures
–ir	All FPGA architectures
–k	All FPGA architectures
–l	All FPGA architectures
–o	All FPGA architectures
–oe	Spartan, SpartanXL, xc4000e/ex/l/xl/xla/xv
–os	Spartan, SpartanXL, xc4000e/ex/l/xl/xla/xv
–p	All FPGA architectures
–pr	All FPGA architectures
–quiet	Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E
–r	All FPGA architectures
–timing	Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E
–tx	Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E
–u	All FPGA architectures

–b (Convert Clock Buffers)

Note This option only applies to XC4000E/L and Spartan.

The –b option replaces GCLKs and ACLKs (primary and secondary clocks) with a generic clock buffer (CKBUF) prior to mapping. This option is useful when you are mapping an XNF netlist created in the Synopsys environment where all clocks are mapped to BUFGP (primary clock buffers) and secondary clocks are not used. The –b gives MAP the greatest amount of latitude in choosing the clock assignments.

–bp (Map Slice Logic)

Note This option only applies to Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E.

The block RAM mapping option is enabled when the –bp option is specified. When block RAM mapping is enabled, MAP attempts to place LUTs and FFs into single-output, single-ported block RAMs.

You can create a file containing a list of register output nets that you want converted into block RAM outputs. To instruct MAP to use this file, set the environment variable XIL_MAP_BRAM_FILE to the file name. MAP looks for this environment variable when the –bp option is specified. Only those output nets listed in the file are made into block RAM outputs.

Note Because block RAM outputs are synchronous and can only be reset, the registers packed into a block RAM must also be synchronous reset.

–c (Pack CLBs)

```
–c [packfactor]
```

The –c option determines the degree to which CLBs are packed when the design is mapped. The valid range of values for the *packfactor* is 0–100.

The *packfactor* values ranging from 1 to 100 roughly specify the percentage of CLBs available in a target device for packing your design's logic.

A *packfactor* of 100 means that all CLBs in a target part are available for design logic. A *packfactor* of 100 results in minimum packing density, while a *packfactor* of 1 represents maximum packing density. Specifying a lower *packfactor* results in a denser design, but the design may then be more difficult to place and route.

The **–c 0** option specifies that only *related* logic (that is, logic having signals in common) should be packed into a single CLB. Specifying **–c 0** yields the least densely packed design.

For values of –c from 1 to 100, MAP merges unrelated logic into the same CLB only if the design requires more resources than are available in the target device (an *overmapped* design). If there are *more* resources available in the target device than are needed by your design, the number of CLBs utilized when **–c 100** is specified may equal the number required when **–c 0** is specified.

Note The **–c 1** setting should only be used to determine the maximum density (minimum area) to which a design can be packed. Xilinx does not recommend using this option in the actual implementation of your design. Designs packed to this maximum density generally have longer run times, severe routing congestion problems in PAR, and poor design performance.

The default *packfactor* (the value if you do not specify a –c option, or enter a –c option without a *packfactor*) is 97% for the XC4000E architecture and 100% for all other XC4000 architectures, Virtex/-E/-II/-II PRO, and Spartan/XL/-II/-IIE.

Processing a design with the **–c 0** option is a good way to get a first estimate of the number of CLBs required by your design.

–cm (Cover Mode)

```
–cm {area | speed | balanced}
```

The –cm option specifies the criteria used during the *cover* phase of MAP. In the this phase, MAP assigns the logic to CLB function generators (LUTs). Use the area, speed, and balanced settings as follows:

- The **area** setting makes reducing the number of LUTs (and therefore the number of CLBs) the highest priority.
- The **speed** setting makes reducing the number of *levels* of LUTS (the number of LUTs a path passes through) the highest priority. This setting makes it easiest to achieve your timing constraints after the design is placed and routed. For most designs there is a small increase in the number of LUTs (compared to the **area** setting), but in some cases the increase may be large.
- The **balanced** setting balances the two priorities—reducing the number of LUTs and reducing the number of levels of LUTs. It produces results similar to the **speed** setting but avoids the possibility of a large increase in the number of LUTs.

The default setting for the –cm option is **area** (cover for minimum number of LUTs). For synthesis-based designs, changing the default does not result in improved performance.

–detail (Write Out Detailed MAP Report)

This option writes out a detailed MAP report. The option replaces the MAP_REPORT_DETAIL environment variable.

–f (Execute Commands File)

```
–f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–fp (Floorplanner)

```
–fp filename.mfp
```

The –fp option requires the specification of an existing MFP file created by the Floorplanner. The MFP file is used as a guide file for mapping.

The MFP file is created in the Floorplanner from a previously mapped NCD file. If you use the –fp option, you cannot use the guide file option (–gf).

For more information about the Floorplanner, see the Floorplanner online help accessible from the Help menu within the Floorplanner.

–gf (Guide NCD File)

```
–gf guidefile
```

The –gf option specifies the name of an existing NCD file (from a previous MAP run) to be used as a guide for the current MAP run. For all Virtex architectures, guided mapping also uses the NGM file. For a description of guided mapping, see the "Guided Mapping" section.

–gm (Guide Mode)

```
–gm {exact | leverage}
```

The –gm option specifies the form of guided mapping to be used**.**

In the EXACT mode the mapping in the guide file is followed exactly. In the LEVERAGE mode, the guide design is used as a starting point for mapping but, in cases where the guided design tools cannot find matches between net and block names in the input and guide designs, or your constraints rule out any matches, the logic is not guided.

For a description of guided mapping, see the "Guided Mapping" section.

–ir (Do Not Use RLOCs to Generate RPMs)

If you enter the –ir option, MAP uses RLOC constraints to group logic within CLBs, but does not use the constraints to generate RPMs (Relationally Placed Macros) controlling the relative placement of CLBs. Stated another way, the RLOCs are not used to control the relative placement of the CLBs with respect to each other.

For the XC4000 and Spartan architectures, the –ir option has an additional behavior; the RLOC constraint that cannot be met is ignored and the mapper will continue processing the design. A warning is generated for each RLOC that is ignored. The resulting mapped design is a valid design.

–k (Map to Input Functions)

The syntax for Spartan-II, Spartan-IIE, Virtex, and Virtex-E architectures follows:

```
–k {4 |5 |6}
```

The syntax for the Virtex-II and Virtex-II PRO architecture follows:

```
–k {4 |5 |6| 7| 8}
```

You can specify the maximum size function that is covered. The default is 4. Covering to 5, 6, 7 or 8 input functions results in the use of F5MUX, F6MUX, and FXMUX.

For XC4000 devices, only –k is specified (not 4, 5 or 6). If the –k option is specified, logic functions of five inputs are mapped into a single CLB (if possible). To perform this mapping, all three of the function generators in the CLB may be used.

By mapping input functions into single CLBs, the –k option may produce a mapping with fewer levels of logic, thus eliminating a number of CLB-to-CLB delays. However, using the –k option may prevent logic from being packed into CLBs in a way that minimizes CLB utilization.

For synthesis-based designs, specifying **–k 4** has little effect. This is because MAP combines smaller input functions into large functions such as F5MUX, F6MUX, F7MUX and F8MUX.

–l (No logic replication)

By default (without the –l option), MAP performs logic replication. Logic replication is an optimization method in which MAP operates on a single driver that is driving multiple loads and maps it as multiple components, each driving a single load (refer to the following figure). Logic replication results in a mapping that often makes it easier to meet your timing requirements, since some delays can be eliminated on critical nets. To turn off logic replication, you must specify the -l option.

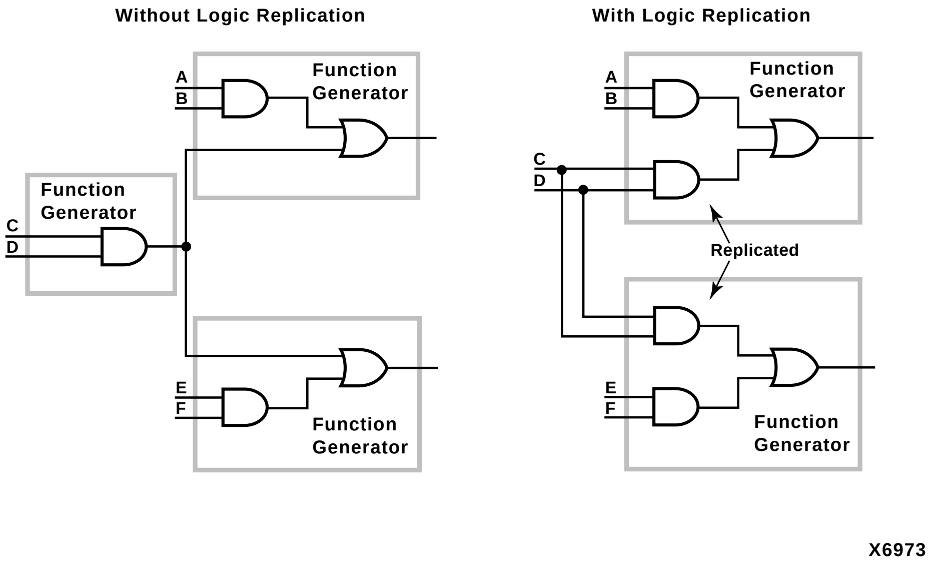


Figure 7-2 Logic Replication (–l Option)

–o (Output File Name)

-o *outfile*[**.ncd**]

Specifies the name of the output NCD file for the design. The .ncd extension is optional. The output file name and its location are determined in the following ways:

- If you do not specify an output file name with the –o option, the output file has the same name as the input file, with an .ncd extension. The file is placed in the input file's directory
- If you specify an output file name with no path specifier (for example, **cpu_dec.ncd** instead of **/home/designs/cpu_dec.ncd)**, the NCD file is placed in the current working directory.
- If you specify an output file name with a full path specifier (for example, **/home/designs/cpu_dec.ncd**), the output file is placed in the specified directory.

If the output file already exists, it is overwritten with the new NCD file. You do not receive a warning when the file is overwritten.

–oe (Logic Optimization Effort)

Note This option only applies to Spartan, SpartanXL, XC4000E / EX / L / XL / XLA / XV.

-oe {**normal** | **high**}

The –oe option specifies the effort MAP uses when performing logic optimization. For the –oe option to apply, the –os (logic optimization style) option must be enabled; that is, –os must have a setting other than **none**. See the "–os (Logic Optimization Style)" section for guidelines on when to use logic optimization.

- If logic optimization is specified by the –os option, the default setting for the –oe option is **normal**.
- In the **high** setting, MAP exerts a greater effort to optimize combinatorial logic, but the mapping takes longer to complete. The **high** setting must be used if the input to the MAP is not optimized, for example, a design created in XABEL.

Note Logic optimization has little effect on most synthesis-based designs.

–os (Logic Optimization Style)

Note This option only applies to Spartan, SpartanXL, XC4000E/EX/L/XL/XLA/XV.

```
–os {area | speed | balanced}
```

Logic optimization in the context of MAP refers to FPGA-specific 4-input lookup optimization by the OPTIX optimizer.

The –os option specifies what type of logic optimization MAP performs.

- The **area** setting optimizes combinatorial logic in a way that uses the minimum number of logic cell function generators (LUTs). This setting minimizes the amount of device area taken up when the design is placed and routed.
- The **speed** setting optimizes in a way that makes it easiest to achieve your timing constraints after the design is placed and routed, even if more function generators must be used.
- The **balanced** setting gives you the optimum combination of area and speed.

The default setting for the –os option disables logic optimization—no optimization is performed. You may want to avoid performing logic optimization in the following cases:

- Your design has already been optimized (for example, a design created in the Synopsys toolset). If the input to MAP has already been optimized (including FPGA-specific 4-input LUT optimization), the MAP results with the –os option enabled may be worse than without the option.
- Your design has been entered as a schematic using Xilinx Unified Library components. In this case, the MAP results with the –os option enabled may be worse than without the option.
- You want to make sure you can perform back-annotation on any of the logic in your original design. Optimization may make some of the logic unavailable for back-annotation.

Note After combinatorial logic optimization has been performed, you lose the correlation between signal names in the NCD file and signal names in the original design. User signal names are not preserved within optimized combinatorial networks. This affects back-annotation and also results in a reduction in the amount of guided mapping and guided placement and routing that can be performed. However, signals connected to pads or to the outputs of BUFTs, flip-flops, latches, and RAMS are preserved for back-annotation.

–p (Part Number)

```
–p part
```

Specifies the Xilinx part number for the device. The syntax for the –p option is described in the "–p (Part Number)" section of the "Introduction" chapter. Examples of *part* entries are **XC4003E-PC84**, and **XC4028EX-HQ240-3**.

If you do not specify a part number using the –p option, MAP selects the part specified in the input NGD file. If the information in the input NGD file does not specify a complete device and package, you must enter a device and package specification using the –p option. MAP supplies a default speed value, if necessary.

The architecture you specify with the –p option must match the architecture specified within the input NGD file. You may have chosen this architecture when you ran NGDBuild or during an earlier step in the design entry process (for example, you may have specified the architecture as an attribute within a schematic, or specified it as an option to a netlist reader). If the architecture does not match, you have to run NGDBuild again and specify the desired architecture.

You can only enter a part number or device name from a device library you have installed on your system. For example, if you have not installed the 4006E device library, you cannot create a design using the 4006E–PC84 part.

–pr (Pack Registers in I/O)

```
–pr {i | o | b}
```

By default (without the –pr option), MAP only places flip-flops or latches within an I/O cell if your design entry method specifies that these components are to be placed within I/O cells. For example, if you create a schematic using IFDX (Input D Flip-Flop) or OFDX (Output D Flip-Flop) design elements, the physical components corresponding to these design elements must be placed in I/O cells. The –pr option specifies that flip-flops or latches may be packed into input registers (**i** selection), output registers (**o** selection), or both (**b** selection) even if the components have not been specified in this way.

–quiet (Report Warnings and Errors Only)

Note This option only applies to Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E.

The –quiet option reduces MAP screen output to warnings and errors only. This option is useful if you only want a summary of the MAP run.

–r (No Register Ordering)

By default (without the –r option), MAP looks at the register bit names for similarities and tries to map register bits in an ordered manner (called *register ordering*). If you specify the -r option, register bit names are ignored when registers are mapped, and the bits are not mapped in any special order. For a description of register ordering, see the "Register Ordering" section.

–timing (Timing-Driven Packing)

Note This option only applies to Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E.

The –timing option directs MAP to give priority to timing critical paths during packing. User-generated timing constraints are used to drive the packing operation.

If you specify this option and one of the following conditions occurs, MAP issues a warning and does not give priority to timing critical paths during packing.

- If the –u option is specified
- If I/O or any other special components are overmapped
- If the mapper runs in any of the guide modes

–tx (Transform Buses)

Note This option only applies to Spartan-II, Spartan-IIE, Virtex, Virtex-II, Virtex-II PRO, Virtex-E.

```
–tx {on | off | aggressive | limit}
```

The –tx option specifies what type of bus transformation MAP performs. The four permitted settings are on, off, aggressive, and limit. The following example shows how the settings are used. In this example, the design has the following characteristics and is mapped to a Virtex device:

- Bus A has 4 BUFTs
- Bus B has 20 BUFTs
- Bus C has 30 BUFTs

MAP processes the design in one of the following ways, based on the setting used for the –tx option:

- The **on** setting performs partial transformation for a long chain that exceeds the device limit.
    - ♦ Bus A is transformed to LUTs (number of BUFTs is >1, ≤4)
    - ♦ Bus B is transformed to CY chain (number of BUFTs is >4, ≤48)
    - ♦ Bus C is *partially* transformed. (25 BUFTs + 1 dummy BUFT due to the maximum width of the XCV50 device + CY chain implementing the other 5 BUFTs)
- The **off** setting turns bus transformation off. This is the default setting.
- The **aggressive** setting transforms the entire bus.
    - ♦ Buses A, B have the same result as the *on* setting.
    - ♦ Bus C is implemented entirely by CY chain. (30 ≤ the default upper limit for carry chain transformation)
- The **limit** setting is the most conservative. It transforms only that portion of the bus that exceeds the device limit.

–u (Do Not Remove Unused Logic)

By default (without the –u option), MAP eliminates unused components and nets from the design before mapping. If –u is specified, MAP maps unused components and nets in the input design and includes them as part of the output design.

The –u option is helpful if you want to run a preliminary mapping on an unfinished design, possibly to see how many components the mapped design uses. By specifying –u, you are assured that all of the design’s logic (even logic that is part of incomplete nets) is mapped.

MAP Process

MAP performs the following steps when mapping a design.

1. Selects the target Xilinx device, package, and speed. MAP selects a part in one of the following ways:
   - ♦ Uses the part specified on the MAP command line.
   - ♦ If a part is not specified on the command line, MAP selects the part specified in the input NGD file. If the information in the input NGD file does not specify a complete architecture, device, and package, MAP issues an error message and stops. If necessary, MAP supplies a default speed.
2. Reads the information in the input design file.
3. Performs a Logical DRC (Design Rule Check) on the input design. If any DRC errors are detected, the MAP run is aborted. If any DRC warnings are detected, the warnings are reported, but MAP continues to run. The Logical DRC (also called the NGD DRC) is described in “Logical Design Rule Check” chapter.

   Note Step 3 is skipped if the NGDBuild DRC was successful.
4. Assigns the device global clock buffers (if possible).
5. Removes unused logic. All unused components and nets are removed, unless the following conditions exist:
   - ♦ A Xilinx S (Save) constraint has been placed on a net during design entry. If an unused net has an S constraint, the net and all used logic connected to the net (as drivers or loads) is retained. All unused logic connected to the net is deleted. For a more complete description of the S constraint, see the *Constraints Guide.*
   - ♦ The –u option was specified on the MAP command line. If this option is specified, all unused logic is kept in the design.
6. Maps pads and their associated logic into IOBs.
7. Maps the logic into Xilinx components (IOBs, CLBs, etc.). If any Xilinx mapping control symbols appear in the design hierarchy of the input file (for example, FMAP or HMAP symbols targeted

to an XC4000EX device), MAP uses the existing mapping of these components in preference to remapping them. The mapping is influenced by various constraints; these constraints are described in the *Constraints Guide*.

8. Update the information received from the input NGD file and write this updated information into an NGM file. This NGM file contains both logical information about the design and physical information about how the design was mapped. The NGM file is used only for back-annotation On Virtex/-E/-II devices, guided mapping uses the NGM file. For more information, see the "Guided Mapping" section.

9. Create a physical constraints (PCF) file. This is a text file containing any constraints specified during design entry. If no constraints were specified during design entry, an empty file is created so that you can enter constraints directly into the file using a text editor or indirectly through the FPGA Editor.

   MAP either creates a PCF file if none exists or rewrites an existing file by overwriting the schematic-generated section of the file (between the statements SCHEMATIC START and SCHEMATIC END). For an existing constraints file, MAP also checks the user-generated section and may either comment out constraints with errors or halt the program. If no errors are found in the user-generated section, the section remains the same.

   Note For Virtex/-E/-II/-II PRO designs, you must use a MAP generated PCF file. The timing tools perform skew checking only with a MAP-generated PCF file.

10. Create an MDF file, which describes how logic was decomposed when the design was mapped. The MDF file is used for guided mapping.

    Note This step does not apply to Virtex/-E/-II/-II PRO or Spartan-II.

11. Run a physical Design Rule Check (DRC) on the mapped design. If DRC errors are found, MAP does not write an NCD file.

12. Create an NCD file, which represents the physical design. The NCD file describes the design in terms of Xilinx components—CLBs, IOBs, etc.

13. Write a MAP report (MRP) file, which lists any errors or warnings found in the design, details how the design was mapped, and supplies statistics about component usage in the mapped design.

Register Ordering

When you run MAP, the default setting performs register ordering. If you specify the –r option, MAP does not perform register ordering and maps the register bits as if they were unrelated.

When you map a design containing registers, the MAP software can optimize the way the registers are grouped into CLBs (slices for Virtex/-E/-II/-II PRO or Spartan-II/-IIE — there are two slices per CLB). This optimized mapping is called *register ordering*.

A CLB (Virtex/-E/-II/-II PRO or Spartan-II/IIE slice) has two flip-flops, so two register bits can be mapped into one CLB. For PAR (Place And Route) to place a register in the most effective way, you want as many pairs of contiguous bits as possible to be mapped together into the same CLBs (for example, bit 0 and bit 1 together in one CLB, bit 2 and bit 3 in another).

MAP pairs register bits (performing *register ordering*) if it recognizes that a series of flip-flops comprise a register. When you create your design, you can name register bits so they are mapped using register ordering.

Note MAP *does not* perform register ordering on any flip-flops which have BLKNM, LOC, or RLOC properties attached to them. The BLKNM, LOC, and RLOC properties define how blocks are to be mapped, and these properties override register ordering.

To be recognized as a candidate for register ordering, the flip-flops must have the following characteristics:

- The flip-flops must share a common clock signal and common control signals (for example, Reset and Clock Enable).
- The flip-flop output signals must all be named according to this convention.
- Output signal names must begin with a common root containing at least one alphabetic character.

The names must end with numeric characters or with numeric characters surrounded by parentheses “( )”, angle brackets “< >”, or square brackets “[ ]”.

For example, acceptable output signal names for register ordering are as follows:

```
data1     addr(04)  bus<1>
data2     addr(08)  bus<2>
data3     addr(12)  bus<3>
data4     addr(16)  bus<4>
                    bus<5>
```

If a series of flip-flops is recognized as a candidate for register ordering, they are paired in CLBs in sequential numerical order. For example, in the first set of names shown above, data1 and data2, are paired in one CLB, while data3 and data4 are paired in another.

In the example below, no register ordering is performed, since the root names for the signals are not identical

data01

addr02

atod03

dtoa04

When it finds a signal with this type of name, MAP ignores the underbar and the numeric characters when it considers the signal for register ordering. For example, if signals are named data00_1 and data01_2, MAP considers them as data00 and data01 for purposes of register ordering. These two signals *are* mapped to the same CLB.

MAP does not change signal names when it checks for underbars—it only ignores the underbar and the number when it checks to see if the signal is a candidate for register ordering.

Because of the way signals are checked, make sure you don’t use an underbar as your bus delimiter. If you name a bus signal data0_01 and a non-bus signal data1, MAP sees them as data0 and data1 and register orders them even though you do not want them register ordered.

Guided Mapping

In guided mapping, an existing NCD is used to guide the current MAP run. The guide file may be from any stage of implementation: unplaced or placed, unrouted or routed. Xilinx recommends that you generate your NCD file using the current release of the software. However, MAP does support guided mapping using NCD files from the previous release.

The following figure shows the guided mapping flow:

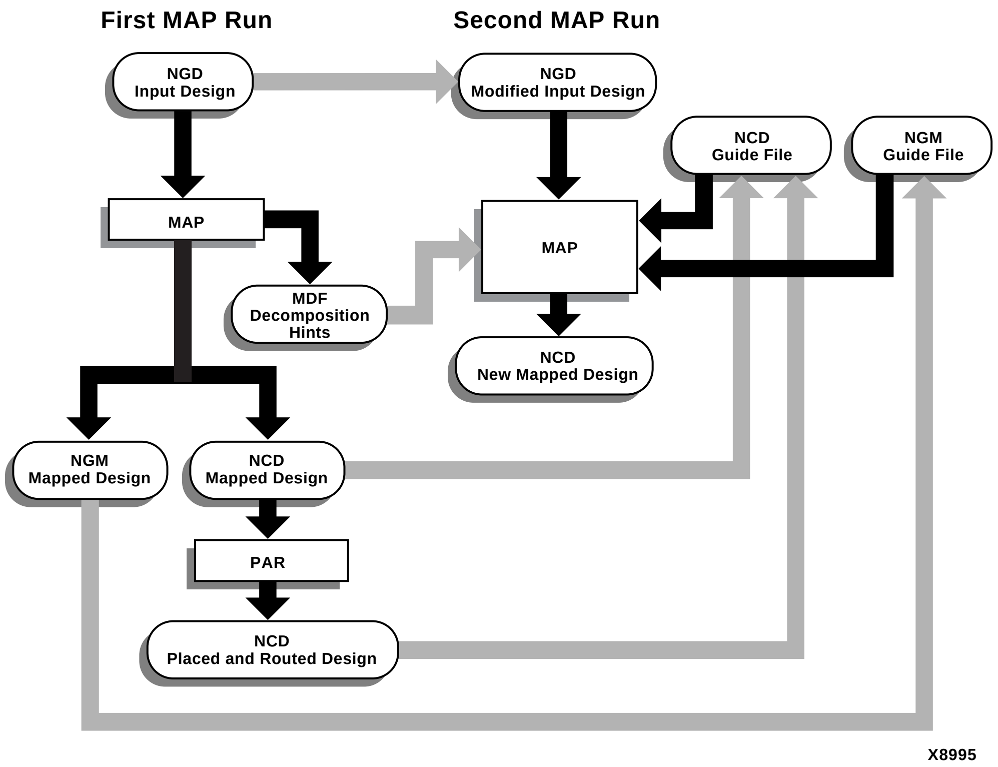


Figure 7-3 Guided Mapping

In the EXACT mode the mapping in the guide file is followed exactly. Any logic in the input NGD file that matches logic mapped into the physical components of the NCD guide file is implemented exactly as in the guide file. Mapping (including signal to pin assignments), placement and routing are all identical. Logic that is not matched to any guide component is mapped by a subsequent mapping step.

If there is a match in EXACT mode, but your constraints would conflict with the mapping in the guide file component, an error is posted. If an error is posted, you can do one of the following:

- Modify the constraints to eliminate conflicts
- Change to the LEVERAGE guide mode (which is less restrictive)
- Modify the logical design changes to avoid conflicts
- Stop using guided design

In the LEVERAGE mode, the guide design is used as a starting point in order to speed up the design process. However, in cases where the guided design tools cannot find matches or your constraints rule out any matches, the logic is not guided. Whenever the guide design conflicts with the your mapping, placement or routing constraints, the guide is ignored and your constraints are followed.

Because the LEVERAGE mode only uses the guide design as a starting point for mapping, MAP may alter the mapping to improve the speed or density of the implementation (for example, MAP may collapse additional gates into a guided CLB).

For Spartan and Virtex/-E/-II/-II PRO devices, MAP uses the NGM and the NCD files as guides. You do not need to specify the NGM file on the command line. MAP infers the appropriate NGM file from the specified NCD file. If MAP does not find an NGM file in the same directory as the NCD, it generates a warning. In this case, MAP uses only the NCD file as the guide file.

Note Guided mapping is not recommended for most HDL designs. Guided mapping depends on signal and component names, and HDL designs often have a low *match rate* when guided. The netlist produced after re-synthesizing HDL modules usually contains signal and instance names that are significantly different from netlists created by earlier synthesis runs. This occurs even if the source level HDL code contains only a few changes.

Simulating Map Results

When simulating with NGM files, you are not simulating a mapped result, you are simulating the logical circuit description. When simulating with NCD files, you are simulating the physical circuit description.

MAP may generate an error that is not detected in the back-annotated simulation netlist. For example, after running MAP, you can run the following command to generate the back-annotated simulation netlist:

```
ngdanno mapped.ncd mapped.ngm –o mapped.nga
```

This command creates a back-annotated simulation netlist using the logical-to-physical cross-reference file named mapped.ngm. This cross-reference file contains information about the logical design netlist, and the back-annotated simulation netlist (mapped.nga) is actually a back-annotated version of the logical design. However, if MAP makes a physical error, for example, implements an Active Low function for an Active High function, this error will not be detected in the mapped.nga file and will not appear in the simulation netlist. For example, consider the following logical circuit generated by NGDBuild from a design file.

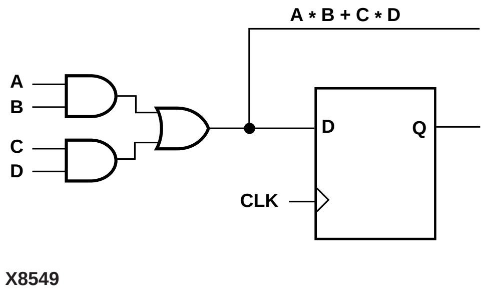


Figure 7-4 Logical Circuit Representation

Observe the Boolean output from the combinatorial logic. Suppose that after running MAP for the preceding circuit, you obtain the following result.

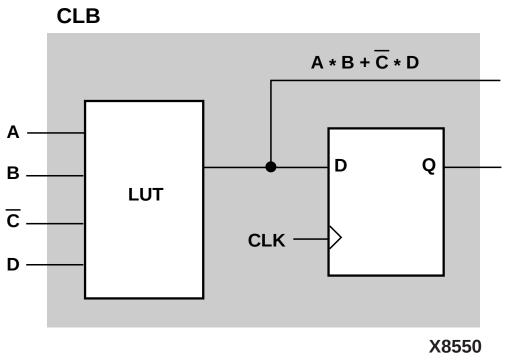


Figure 7-5 CLB Configuration

Observe that MAP has generated an active low (C) instead of an active high (C). Consequently, the Boolean output for the combinatorial logic is incorrect. When you run NGDAnno using the mapped.ngm file (ngdanno mapped.ncd mapped.ngm –o mapped.nga), you cannot detect the logical error because the delays are back-annotated to the correct logical design, and not to the physical design.

One way to detect the error is by running the NGDAnno command without using the mapped.ngm cross-reference file.

```
ngdanno mapped.ncd –o mapped.nga
```

As a result, physical simulations using the mapped.nga file should detect a physical error. However, the type of error is not always easily recognizable. To pinpoint the error, use the FPGA Editor or call Xilinx Customer Support. In some cases, a reported error may not really exist, and the CLB configuration is actually correct. You can use the FPGA Editor to determine if the CLB is correctly modelled.

Finally, if both the logical and physical simulations do not discover existing errors, you may need to use more test vectors in the simulations.

MAP Report (MRP) File

The MAP report (MRP) file is an ASCII text file that contains information about the MAP run. The report information varies depending on the device and whether you use the -detail option (see the "–detail (Write Out Detailed MAP Report)" section).

An abbreviated MRP file is shown below—most report files are considerably larger than the one shown. The file is divided into a number of sections, and sections appear even if they are empty. The sections of the MRP file are as follows:

- Design Information—Shows your MAP command, line, the device to which the design has been mapped, and when the mapping was performed.
- Design Summary—Summarizes the mapper run, showing the number of errors and warnings, and how many of the resources in the target device are used by the mapped design.
- Table of Contents—Lists the remaining sections of the MAP report.
- Errors—Shows any errors generated as a result of the following:
    - ♦ Errors associated with the logical DRC tests performed at the beginning of the mapper run. These errors do not depend on the device to which you are mapping.
    - ♦ Errors the mapper discovers (for example, a pad is not connected to any logic, or a bidirectional pad is placed in the design but signals only pass in one direction through the pad). These errors may depend on the device to which you are mapping.
    - ♦ Errors associated with the physical DRC run on the mapped design.
- Warnings—Shows any warnings generated as a result of the following:
    - ♦ Warnings associated with the logical DRC tests performed at the beginning of the mapper run. These warnings do not depend on the device to which you are mapping.
    - ♦ Warnings the mapper discovers. These warnings may depend on the device to which you are mapping.

    - Warnings associated with the physical DRC run on the mapped design.

- Informational—Shows messages that usually do not require user intervention to prevent a problem later in the flow. These messages contain information that may be valuable later if problems do occur.
- Removed Logic Summary—Summarizes the number of blocks and signals removed from the design. The section reports on these kinds of removed logic.
- Blocks trimmed—A trimmed block is removed because it is along a path that has no driver or no load. Trimming is recursive. For example, if Block A becomes unnecessary because logic to which it is connected has been trimmed, then Block A is also trimmed.
    - Blocks removed—A block is removed because it can be eliminated without changing the operation of the design. Removal is recursive. For example, if Block A becomes unnecessary because logic to which it is connected has been removed, then Block A is also removed.
    - Blocks optimized—An optimized block is removed because its output remains constant regardless of the state of the inputs (for example, an AND gate with one input tied to ground). Logic generating an input to this optimized block (and to no other blocks) is also removed, and appears in this section.
    - Signals removed—Signals are removed if they are attached only to removed blocks.
    - Signals merged—Signals are merged when a component separating them is removed.
- Removed Logic—Describes in detail all logic (design components and nets) removed from the input NGD file when the design was mapped. Generally, logic is removed for the following reasons:
    - The design uses only part of the logic in a library macro.
    - The design has been mapped even though it is not yet complete.
    - The mapper has optimized the design logic.

  - ♦ Unused logic has been created in error during schematic entry.

  This section also indicates which nets were merged (for example, two nets were combined when a component separating them was removed).

  In this section, if the removal of a signal or symbol results in the subsequent removal of an additional signal or symbol, the line describing the subsequent removal is indented. This indentation is repeated as a chain of related logic is removed. To quickly locate the cause for the removal of a chain of logic, look above the entry in which you are interested and locate the top-level line, which is not indented.

- IOB Properties—Lists each IOB to which the user has supplied constraints along with the applicable constraints.
- RPMs—Indicates each RPM (Relationally Placed Macro) used in the design, and the number of device components used to implement the RPM.
- Guide Report—If you have mapped using a guide file, shows the guide mode used (EXACT or LEVERAGE) and the percentage of objects that were successfully guided.
- Area Group Summary—The mapper summarizes results for each area group. MAP uses area groups to specify a group of logical blocks that are packed into separate physical areas.
- Modular Design Summary—After the Modular Design Active Module Implementation Phase, this section lists the logic that was added to the design to successfully implement the active module. After the Final Assembly Phase, this section states whether the logic was assembled successfully.

Note The MAP Report is formatted for viewing in a monospace (non-proportional) font. If the text editor you use for viewing the report uses a proportional font, the columns in the report do not line up correctly.

```
Xilinx Mapping Report File for Design 'udcntr'

Design Information
------------------
Command Line   : map udcntr.ngd -o udcntr_map.ncd
Target Device  : xv300
Target Package : bg432
Target Speed   : -5
Mapper Version : virtex -- $Revision: 1.58 $
Mapped Date    : Wed May 23 10:32:53 2001

Design Summary
--------------
   Number of errors:      0
   Number of warnings:    1
   Number of Slices:                    3 out of  3,072    1%
   Number of Slices containing
      unrelated logic:                  0 out of      3    0%
   Number of Slice Flip Flops:          4 out of  6,144    1%
   Number of 4 input LUTs:              6 out of  6,144    1%
   Number of bonded IOBs:              18 out of    316    5%
   Number of Tbufs:                     8 out of  3,200    1%
   Number of GCLKs:                     1 out of      4   25%
   Number of GCLKIOBs:                  1 out of      4   25%
   Number of hard macros:             1
Total equivalent gate count for design (not including hard macros):
68
Additional JTAG gate count for IOBs:  912

Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Informational
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - IOB Properties
Section 7 - RPMs
Section 8 - Guide Report
Section 9 - Area Group Summary
Section 10 - Modular Design Summary
Section 1 - Errors
------------------

Section 2 - Warnings
--------------------
WARNING:MapLib:328 - Block U2 is not a recognized logical block. The
mapper will continue to process the design but there may be design
problems if this block does not get trimmed.

Section 3 - Informational
-------------------------
INFO:MapLib:62 - All of the external outputs in this design are
using slew rate limited output drivers. The delay on speed critical
outputs can be dramatically reduced by designating them as fast
outputs in the schematic.

Section 4 - Removed Logic Summary
---------------------------------
   3 block(s) removed
   1 block(s) optimized away
   3 signal(s) removed

Section 5 - Removed Logic
-------------------------

The trimmed logic report below shows the logic removed from your
design due to sourceless or loadless signals, and VCC or ground
connections.  If the removal of a signal or symbol results in the
subsequent removal of an additional signal or symbol, the message
explaining that second removal will be indented.  This indentation
will be repeated as a chain of related logic is removed.

To quickly locate the original cause for the removal of a chain of
logic, look above the place where that logic is listed in the
trimming report, then locate the lines that are least indented
(begin at the leftmost edge).

The signal "VCC" is loadless and has been removed.
 Loadless block "VCC" (ONE) removed.
The signal "U1/GND" is sourceless and has been removed.
The signal "U1/VCC" is sourceless and has been removed.
Unused block "U1/GND" (ZERO) removed.
Unused block "U1/VCC" (ONE) removed.

Optimized Block(s):
TYPE            BLOCK
GND             GND

Section 6 - IOB Properties
--------------------------


 IOB     Type     Direc-  IO      Drive    Slew  Reg(s)  Registor  IOB
 Name             tion    Stan-   Strength Rate                    Delay
                          dard
 clock   GCLKIOB  Input   LVTTL
 IN[0]   IOB      Input   LVTTL
 IN[1]   IOB      Input   LVTTL
 IN[1]   IOB      Input   LVTTL
 IN[3]   IOB      Input   LVTTL
 Q1[0]   IOB      Output  LVTTL   12       SLOW
 Q1[1]   IOB      Output  LVTTL   12       SLOW
 Q1[2]   IOB      Output  LVTTL   12       SLOW
 Q1[3]   IOB      Output  LVTTL   12       SLOW
 Q2[0]   IOB      Output  LVTTL   12       SLOW
 Q2[1]   IOB      Output  LVTTL   12       SLOW
 Q2[2]   IOB      Output  LVTTL   12       SLOW
 Q2[3]   IOB      Output  LVTTL   12       SLOW
 clear1  IOB      Input   LVTTL
 clear2  IOB      Input   LVTTL
 load1   IOB      Input   LVTTL
 load2   IOB      Input   LVTTL
 triL    IOB      Input   LVTTL
 triR    IOB      Input   LVTTL
```

```
Section 7 - RPMs
----------------

Section 8 - Guide Report
------------------------
Guide not run on this design.

Section 9 - Area Group Summary
------------------------------
   AREA_GROUP AG_U1
   RANGE: CLB_R1C1.*:CLB_R32C24.*
   No COMPRESSION specified for AREA_GROUP AG_U1
      Number of Slices:              3 out of  1,536    1%
      Number of Slice Flip Flops:    4 out of  3,072    1%
      Total Number 4 input LUTs:     6 out of  3,072    1%
         Number used as 4 input LUTs:              6


Section 10 - Modular Design Summary
-----------------------------------
The following logic was added to the design to satisfy the
active module’s interface.  These interface components will
be removed during the Modular Design Final Assembly Phase.

  0 Flip Flops.
  0 LUTs
  0 TBUFs

To get a listing of the active module port nets, set the
"XIL_MAP_LISTPORTNETS" environment variable and rerun map.
```

Halting MAP

To halt MAP, enter Ctrl C (on a workstation) or Ctrl-break (on a PC). On a workstation, make sure that when you enter Ctrl C the active window is the window from which you invoked the mapper. The operation in progress is halted. Some files may be left when the mapper is halted (for example, a MAP report file or a physical constraints file), but these files may be discarded since they represent an incomplete operation.

Chapter 8

Physical Design Rule Check

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA

The chapter contains the following sections:

- "DRC Overview"
- "DRC Syntax"
- "DRC Input File"
- "DRC Output File"
- "DRC Options"
- "DRC Checks"
- "DRC Errors and Warnings"

DRC Overview

The physical Design Rule Check, also known as DRC, comprises a series of tests to discover physical errors and some logic errors in the design. The physical DRC is run as follows:

- MAP automatically runs physical DRC after it has mapped the design.
- PAR (Place and Route) automatically runs physical DRC on nets when it routes the design.
- BitGen, which creates a a BIT file for programming the device, automatically runs physical DRC.

- You can run physical DRC from within the FPGA Editor. The DRC also runs automatically after certain FPGA Editor operations (for example, when you edit a logic cell or when you manually route a net). For a description of how the DRC works within the FPGA Editor, see the online Help provided with this GUI.
- You can run physical DRC from the UNIX or DOS command line.

DRC Syntax

The following command runs physical DRC:

```
drc [options] file_name
```

options can be any number of the DRC options listed in the "DRC Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

file_name is the name of the NCD file on which DRC is to be run.

DRC Input File

The input to DRC is an NCD file. The NCD file is a mapped, physical description of your design.

DRC Output File

The output of DRC is a TDR file. The TDR file is an ASCII DRC report. The contents of this file are determined by the options you select for the DRC command.

DRC Options

This section describes the DRC command line options.

–e (Error Report)

The –e option produces a report containing details about errors only. No details are given about warnings.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–o (Output file)

```
-o outfile_name
```

The –o option overrides the default output report file *file_name*.tdr with *outfile_name*.tdr.

–s (Summary Report)

The –s option produces a summary report only. The report lists the number of errors and warnings found but does not supply any details about them.

–v (Verbose Report)

The –v option reports all warnings and errors. This is the default option for DRC.

–z (Report Incomplete Programming)

The –z option reports incomplete programming as errors. Certain DRC violations are considered errors when the DRC runs as part of the BitGen command but are considered warnings at all other times the DRC runs. These violations usually indicate the design is incompletely programmed (for example, a logic cell has been only partially programmed or a signal has no driver). The violations create errors if you try to program the device, so they are reported as errors when BitGen creates a BIT file for device programming. If you run DRC from the command line without the –z option, these violations are reported as warnings only. With the –z option, these violations are reported as errors.

DRC Checks

Physical DRC performs the following types of checks:

- Net check

  This check examines one or more routed or unrouted signals and reports any problems with pin counts, 3-state buffer inconsistencies, floating segments, antennae, and partial routes.

- Block check

  This check examines one or more placed or unplaced components and reports any problems with logic, physical pin connections, or programming.

- Chip check

  This check examines a special class of checks for signals, components, or both at the chip level, such as placement rules with respect to one side of the device.

- All checks

  This check performs net, block, and chip checks.

When you run DRC from the command line, it automatically performs net, block, and chip checks.

In the FPGA Editor, you can run the net check on selected objects or on all of the signals in the design. Similarly, the block check can be performed on selected components or on all of the design’s components. When you check all components in the design, the block check performs extra tests on the design as a whole (for example, 3-state buffers sharing long lines and oscillator circuitry configured correctly) in addition to checking the individual components. In the FPGA Editor, you can run the net check and block check separately or together.

DRC Errors and Warnings

A DRC error indicates a condition in which the routing or component logic does not operate correctly (for example, a net without a driver or a logic block that is incorrectly programmed). A DRC warning indicates a condition where the routing or logic is incomplete (for example, a net is not fully routed or a logic block has been programmed to process a signal but there is no signal on the appropriate logic block pin).

Certain messages may appear as either warnings or errors, depending on the application and signal connections. For example, in a net check, a pull-up not used on a signal connected to a decoder generates an error message. A pull-up not used on a signal connected to a 3-state buffer only generates a warning.

Incomplete programing (for example, a signal without a driver or a partially programmed logic cell) is reported as an error when the DRC runs as part of the BitGen command, but is reported as a warning when the DRC runs as part of any other program. The –z option to the DRC command reports incomplete programming as an error instead of a warning. For a description of the –z option, see the "–z (Report Incomplete Programming)" section.

Chapter 9

PAR

This program is compatible with the following families:

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA

The chapter contains the following sections:

PAR Overview

After you create a mapped NCD (Native Circuit Description) file, you can place and route the file using PAR. PAR accepts an NCD file as input, places and routes the design, and outputs an NCD file to be used by the bitstream generator (BitGen). You can use the output NCD file as a guide file for additional runs of PAR after making minor changes to your design.

PAR places and routes a design based on the following considerations:

- Cost-based—Placement and routing are performed using various cost tables that assign weighted values to relevant factors such as constraints, length of connection, and available routing resources. Cost-based placement and cost-based routing are described in the "PAR Process" section.
- Timing-Driven—The Xilinx timing analysis software enables PAR to place and route a design based upon your timing constraints (see the "Timing-driven PAR" section).

The design flow through the PAR module is shown in the following figure. This figure shows a PAR run that produces a single output design file.

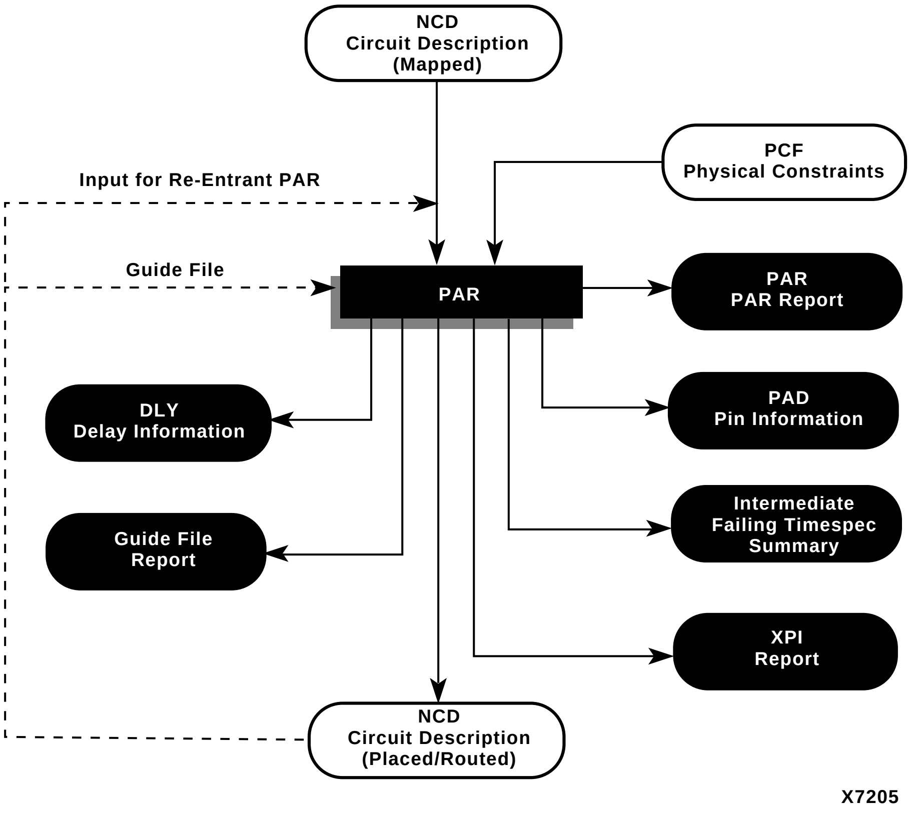


Figure 9-1 PAR Flow

PAR Syntax

The following syntax places and routes your design.

par [*options*] *infile*[**.ncd**] *outfile pcf_file*[**.pcf**]

options can be any number of the PAR options listed in the "PAR Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

infile is the design file you wish to place and / or route. The file must include an .ncd extension, but you do not have to specify the .ncd extension on the command line.

outfile is the target design file that is written after PAR is finished. If the command options you specify yield a single output design file, *outfile* has an extension of .ncd or .dir. An .ncd extension generates an output file in NCD format, and the .dir extension directs PAR to create a directory in which to place the output file (in NCD format). If the specified command options yield more than one output design file, *outfile* must have an extension of .dir. The multiple output files (in NCD format) are placed in the directory with the .dir extension.

If the file or directory you specify already exists, you get an error message and the operation does not run. You can override this protection and automatically overwrite existing files by using the –w option.

pcf_file is a physical constraints file. The file contains the constraints you entered during design entry, constraints you added using the UCF (User Constraints File), and constraints you added directly in the PCF file. If you do not enter the name of a physical constraints file on the command line and the current directory contains an existing physical constraints file with the *infile* name and a .pcf extension, PAR uses the PCF file.

PAR Input Files

Input to PAR consists of the following files:

- NCD file—a mapped design.
- PCF file—an ASCII file containing constraints based on attributes in the schematic or on constraints placed in a UCF file. A complete list of constraints can be found in the *Constraints Guide.* PAR supports all of the timing constraints described in that manual.
- Guide NCD file—an optional placed and routed NCD file you can use as a guide for placing and routing the design.

PAR Output Files

Output from PAR consists of the following files:

- NCD file—a placed and routed design file (may contain placement and routing information in varying degrees of completion).

- PAR file—a PAR report including summary information of all placement and routing iterations.
- DLY file—a file containing delay information for each net in the design.
- PAD file—a file containing I/O pin assignments.
- GRF (Guide Report File)— a file that is created when you use the –gf option.
- ITR (Intermediate Timespec Report)—a file containing a summary of failing timing specifications.
- XPI (Xilinx Par Information)—a file containing information on whether or not the design routed and if timing specifications were met.

PAR Options

You can customize the place and route operation by specifying options when you run PAR. You can place a design without routing it, specify the maximum number of passes the router performs, specify the number and type of cleanup passes the router runs, perform a single placement, perform a number of placements using different cost tables, and specify an effort level to indicate whether the design is simple or complex.

The following table is a summary of PAR options, default settings, and setting ranges.

Option	Default Setting	Range
–c *number*	Default is 0 for Spartan-II, Virtex/-E/-II Default is 1 for all other devices	0–5
–d *number*	0	0–5
–detail	No detailed PAR report	N/A
–dfs	Run connection-based method (No –dfs, –kpaths is the default)	N/A
–e *number*	0 (No delay-based router cleanup passes on completely routed designs)	0–5
–f *command_file*	No command line file	N/A

Option	Default Setting	Range
–gf	No guide file	N/A
–gm [**leverage** | **exact**]	Exact	N/A
–guide *guide_file* [**leverage** | **exact**]	Exact	N/A
–i *number*	Run until completion or until router decides it can not complete	0–2000
–k	Run placement (do not run re-entrant routing)	N/A
–**kpaths**	Run connection-based method (–kpaths is the default)	N/A
–l *number*	2 (Overall effort level 2)	1–5
–m *nodefile_name*	Do not run the Turns Engine	N/A
–n *number*	1 (One place and route iteration)	0–100
–ol *number*	2 (Overall effort level 2)	1–5
–p	Run placement	N/A
–pl *number*	Determined by –ol setting	1–5
–r	Run router	N/A
–rl *number*	Determined by –ol setting	1–5
–s *number*	Save all	1–100
–t *number*	1 (Start placer at cost table 1)	1–100
–ub	Do not use bonded I/Os	N/A
–w	Do not overwrite	N/A
–x	Use timing constraints in PCF file	N/A
–xe *effort_level*	Set the extra effort level	0-5

–c (Number of Cost-Based Router Cleanup Passes)

```
–c cost_passes
```

There are two types of cleanup routing you can perform—a faster cost-based cleanup routing and a more intensive delay-based cleanup routing. Cost-based cleanup runs much faster than delay-based cleanup, but delay-base cleanup usually produces a result that has faster in-circuit performance. The valid range of *cost_passes* is 0–5. The default setting for –c is 1 for all devices except Spartan-II and Virtex/-E/-II, for which the default is 0.

Following are reasons to use the cost-based cleanup passes:

- For non-timing driven runs, cleanup routing can significantly improve delays.
- For timing-driven runs, the cleanup passes can improve timing on those elements of the design that are not covered by timing constraints.

Note The –c option is not recommended for use with Virtex/-E/-II or Spartan-II. This option creates longer runtimes with little improvement.

–d (Number of Delay-Based Router Cleanup Passes)

```
–d delay_passes
```

By default (without the –d option), the router does not run any delay-based cleanup passes (described in the "Routing" section). If you run both delay-based cleanup passes and cost-based cleanup passes (see –c option above), the cost-based passes run before the delay-based passes. Typically, the first delay-based cleanup pass produces the greatest improvement, with less improvement on each successive pass. It is also possible that delay passes do not show any improvement. The valid range of *delay_passes* is 0–5, and the default is 0.

If you want to run delay-based cleanup passes only on designs that are routed to completion (100% routed), use the –e option instead of the –d option.

Note The –d option is not recommended for use with Virtex/-E/II or Spartan-II. The evaluation of this option with these architectures indicates that the option creates much longer runtimes with little improvement.

–detail (Write Out Detailed PAR Report)

This option writes out a detailed PAR report. The option replaces the PAR_REPORT_DETAIL environment variable.

–dfs (Thorough Timing Analysis of Paths)

The –dfs option specifies that PAR utilize depth-first search timing analysis, which analyzes all paths covered by timing constraints in order to perform timing-driven place and route. This method is more thorough than the default method (–kpaths) and may result in longer PAR runtimes. See the "–kpaths (Faster Analysis of Paths)" section for a discussion of the connection-based method.

–e (Delay-based Cleanup Passes—Completely Routed Designs)

```
-e number
```

The –e option operates in the same way as the –d option, however, the –d option runs on *all* output designs produced by the PAR run, while the –e option only runs on those output designs that are routed to completion. The *number* of passes is 0–5, and the default is 0.

Note This option is not recommended for use with Virtex/-E/-II or Spartan-II. The evaluation of this option with these architectures indicates that the option creates much longer runtimes with little improvement.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–gf (Guide NCD File)

```
-gf guide_file
```

The –gf option specifies the name of an NCD file (from a previous MAP or PAR run) to be used as a guide for this PAR run. The guide file is an NCD file which is used as a template for placing and routing the input design. For more information on the guide file, see the "Guided PAR" section.

–gm (Guide Mode)

```
-gm {exact | leverage}
```

The –gm option specifies the type of guided placement and routing PAR uses—exact or leveraged. The default is exact mode. For more information on the guide modes, see the "Guided PAR" section.

You specify the NCD to use as a guide file by entering a –gf option (see the "–gf (Guide NCD File)" section) on the PAR command line. If you do not specify a guide file, PAR is guided by the placement and routing information in the input NCD file. The "Guided PAR" section describes how PAR operates if no guide file is specified.

–guide (Guide File)

–guide *guide_file* **{exact | leverage}**

Use the –guide option when you are using multiple guide files, or as an alternative to the –gm or –gf option. Use the *guide_file* variable to specify the names of the NCD files that you want to use as guides for your current PAR run.

The *exact* and *leverage* modes specify the form of the guided placement and routing. If a guide mode is not entered, then the value specified for –gm is used or the default (*exact*) is used.

You can use the –guide option to override the guide mode for Modular Design. For more information on the guide modes, see the "Guided PAR" section.

–i (Number of Router Iterations)

–i *route_passes*

By default (without the –i option), the router runs until it either routes to 100% completion and meets its timing constraints, or intelligently determines that it cannot succeed. When you specify the –i option, the router runs a maximum number of passes, stopping earlier only if the routing goes to 100% completion and all constraints are met. Each pass is a single attempt to route a placement to completion, and the screen displays a message for each pass. The valid range of *route_passes* is 0–2000.

–k (Re-Entrant Routing)

The –k option runs re-entrant (also called incremental) routing. Routing begins with the existing placement and routing left in place. Re-entrant routing is useful if you wish to manually route parts of the design and then continue automatic routing, if you halted the route prematurely (for example, with a Ctrl C) and wish to resume, or if you wish to run additional route or delay reduction passes.

–kpaths (Faster Analysis of Paths)

The –kpaths option runs the connection-based method of path analysis (this is the default). The non-enumerative connection-based method has a runtime proportional to the size of the design, unlike the DFS method, which has a runtime proportional to the number of paths in the design.

There are two significant differences between the connection-based method and the DFS method.

- The DFS method analyzes all paths except those that actually contain a circuit cycle, including paths that contain connections that cause a circuit cycle for other paths in the circuit. The connection-based method may not analyze these paths depending on circuit topology, as shown in the following example.

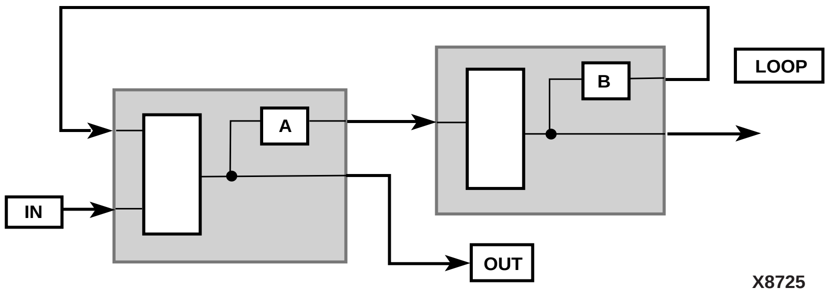


Figure 9-2 Circuit Cycles

The DFS method traces the path from IN, through A, through the signal LOOP, back to the left-most logic block and to the signal OUT. The new connection-based method may not trace this path because a combinatorial cycle exists at the output of A.

- The DFS method removes false paths from a design that requires contending 3-state enable signals. The connection-based method does not perform this optimization which means that it may analyze some paths that are statically false based on 3-state enable signals, as shown in the following example.

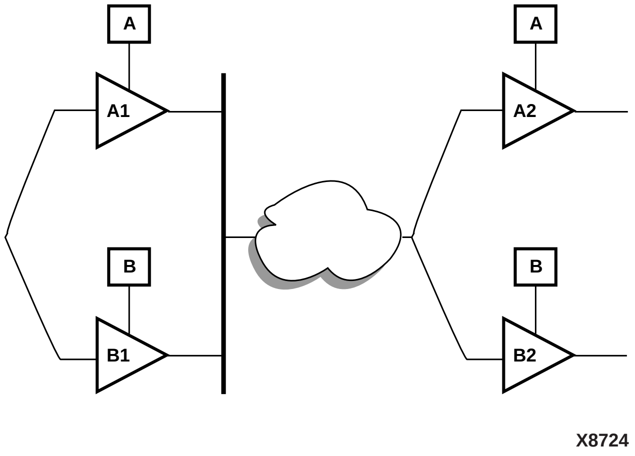


Figure 9-3 3-state Buffer Paths

A signal can pass through four paths in the preceding circuit but two of the paths are false (A1 to B2 and B1 to A2). In order for a signal to pass through the upper left 3-state buffer A1, the enable signal A must be true. In order to prevent a bus contention on the A1 output, the enable signal B must be false. Since buffer B2 is also controlled by the enable signal B, the path through A1 cannot pass through B2 (because when A is enabled, B is disabled). The converse is also true, if B is enabled, the only valid path is from B1 to B2.

In the example circuit with the DFS method, only true paths are considered. The connection-based method traces the false paths as well as the true paths.

–l (Overall Effort Level)

```
-l effort_level
```

The –l option is identical to the –ol option. See the "–ol (Overall Effort Level)" section.

–m (Multi-Tasking Mode)

```
-m nodefile_name
```

The –m option allows you to specify the nodes on which to run jobs when using the PAR Turns Engine. You must use this option with the -n (Number of PAR Iterations) option.

–n (Number of PAR Iterations)

```
-n iterations
```

By default (without the –n option), one place and route iteration is run. The –n option determines the number of iterations (place and route passes) performed at the effort level specified by the –l option. Each iteration uses a different cost table when the design is placed and produces a different NCD file. If you enter -n 0, the software continues to place and route, stopping either after the design is fully routed or after completing the iteration at cost table 100 and meeting all timing constraints. If you specify a –t option, the iterations begin at the cost table specified by –t. The valid range of *iterations* is 0–100, and the default is 1.

–ol (Overall Effort Level)

```
-ol effort_level
```

The –ol option sets the overall PAR effort level. The effort level specifies the level of effort PAR uses to place and route your design to completion and achieve your timing constraints.

There are five values for *effort_level*. Level 1 should be used on the least complex design, and level 5 should be used on the most complex. The level is not an absolute; it shows instead relative effort.

If you place and route a simple design at a complex level, the design is placed and routed properly, but the process takes more time than placing and routing at a simpler level. If you place and route a complex design at a simple level, the design may not route to completion or may route less completely (or with worse delay characteristics) than at a more complex level.

The *effort_level* range is 1–5, and the default level is 2.

The –ol level sets an effort level for placement and another effort level for routing. These levels also range from 1–5. The placement and routing levels set at a given –ol level depend on the device family in the NCD file. You can determine the default placer and router effort levels for a device family by reading the PAR Report file produced by your PAR run.

You can override the placer level set by the –ol option by entering a –pl (Placer Effort Level) option, and you can override the router level by entering a –rl (Router Effort Level) option.

Note For Spartan-II and Virtex/-E devices, automatic timespecing is performed if PAR does not detect timing constraints and the effort level is set at 3, 4, or 5. See the"Automatic Timespecing" section in this chapter for more information.

–p (No Placement)

The –p option bypasses both constructive and optimizing placement (described in the "Placing" section). When you use this option, existing routes are ripped up before routing begins. You can, however, leave the existing routing in place if you use the –k option instead of the –p option.

–pl (Placer Effort Level)

```
-pl placer_effort_level
```

The –pl option sets the placer effort level. The effort level specifies the level of effort used when placing the design. Level 1 should be used on the least complex design, and level 5 should be used on the most complex. For a description of effort level, see the "–ol (Overall Effort Level)" section.

The *placer_effort_level* range is 1–5, and the default level set if you do not enter a –pl option is determined by the setting of the –ol option. This default varies depending on the device family in the input NCD file. You can determine the default placer effort level for a given –ol level and device family by reading the PAR Report file produced by your PAR run.

–r (No Routing)

Use the –r option to prevent the routing of a design.

–rl (Router Effort Level)

```
-rl router_effort_level
```

The –rl option sets the router effort level. The effort level specifies the level of effort used when routing the design. Level 1 should be used on the least complex design, and level 5 should be used on the most complex. For a description of effort level, see the "–ol (Overall Effort Level)" section.

The *router_effort_level* range is 1–5, and the default level set if you do not enter a –rl option is determined by the setting of the –ol option. This default varies depending on the device family in the input NCD file. You can determine the default router effort level for a given –ol level and device family by reading the PAR Report file produced by your PAR run.

–s (Number of Results to Save)

```
-s number_to_save
```

By default (without the -s option), all results are saved. The –s option saves only the number of results you specify. The –s option compares every result to every other result and leaves you with the best number of NCD files. The best outputs are determined by a score assigned to each output design. This score takes into account such factors as the number of unrouted nets, the delays on nets and conformance to your timing constraints. The lower the score, the better the design. This score is described in the "PAR Reports" section. The valid range for *number_to_save* is 0–100, and the default –s setting (no –s option specified) saves all results.

–t (Starting Placer Cost Table)

```
-t placer_cost_table
```

By default (without the -t option), PAR starts at placer cost table 1.The –t option specifies the cost table at which the placer starts (placer cost tables are described in the "Placing" section). If cost table 100 is reached, placement does not begin at 1 again, even if command options specify that more placements should be performed. The *placer_cost_table range* is 1–100, and the default is 1.

–ub (Use Bonded I/Os)

By default (without the -ub option), I/O logic that MAP has identified as internal can only be placed in unbonded I/O sites. If the –ub option is specified, PAR can place this internal I/O logic into bonded I/O sites in which the I/O pad is not used. The option also allows PAR to route through bonded I/O sites. If you use the –ub option, make sure this logic is not placed in bonded sites connected to external signals, power, or ground. You can prevent this condition by placing PROHIBIT constraints on the appropriate bonded I/O sites. See the *Constraints Guide* for more information on constraints.

–w (Overwrite Existing Files)

Use the –w option to instruct PAR to overwrite existing files, including the input design file.

–x (Ignore Timing Constraints)

If you do not specify the –x option, the PAR software automatically runs a timing-driven PAR run if any timing constraints are found in the physical constraints file. If you do specify –x, timing-driven PAR is not invoked in any case.

The –x option might be used if you have timing constraints specified in your physical constraints file, but you want to execute a quick PAR run without using the timing-driven PAR feature, to give you a rough idea of how difficult the design is to place and route.

Note On Spartan-II and Virtex/-E devices, using the -x option suppresses the automatic timespecing feature. For more information, see the "Automatic Timespecing" section in this chapter.

–xe (Extra Effort Level)

```
–xe effort_level
```

Use the –xe option to set the extra effort level. The effort_level variable can equal any number between 0 and 5. Use level 0 to turn off all extra effort processing.

PAR Process

This section provides information on how placing and routing are performed by PAR, as well as information on timing-driven PAR and automatic timespecing.

Placing

The PAR module places in two stages: a constructive placement and an optimizing placement. PAR writes the NCD file after constructive placement and modifies the NCD after optimizing placement.

During constructive placement, PAR places components into sites based on factors such as constraints specified in the input file (for example, certain components must be in certain locations), the length of connections, and the available routing resources. This placement also uses *cost tables*. Cost tables assign weighted values to relevant factors such as constraints specified in the input file (for example, certain components must be in certain locations), the length of connections, and the available routing resources. You can specify a cost table value from 1 to 100. The number you select corresponds to a cost table index and results in different place and route strategies. The default value is 1. Constructive placement continues until all components are placed. PAR writes the NCD file after constructive placement.

The optimizing placement is a fine tuning of the results of the constructive placement. Optimizing is run only at specific levels, and the number of passes may vary. PAR rewrites the NCD file after optimizing placement.

Timing-driven placement is automatically invoked if PAR finds timing constraints in the physical constraints file.

Routing

Routing is done in two stages: constructive routing and cleanup. PAR writes the NCD file only at the end of an iteration after more than 60 minutes of routing have elapsed, and it only writes out a new NCD file if the design quality improves.

During constructive routing, the router performs an iterative procedure to converge on a solution that routes the design to completion and meets timing constraints. If both of these goals cannot be met, the first is considered more important, and PAR tries to route to completion even if timing constraints are not met.

During cleanup routing, the router takes the result of constructive routing and reroutes some connections to minimize the delays on all nets and decrease the number of routing resources used. If both of these goals cannot be met, the first is considered more important, and PAR tries to route to minimize delays over decreasing the number of routing resources used.

There are two types of cleanup routing you can perform—a faster cost-based cleanup routing and a more intensive delay-based cleanup routing. Cost-based cleanup runs much faster than delay-based cleanup, but delay-base cleanup usually produces a result that has faster in-circuit performance.

Timing-driven routing is automatically invoked if PAR finds timing constraints in the physical constraints file.

Note To achieve your timing constraints while routing an XC4000E/L/EX/XL design, PAR may add an additional pull-up to a net at the output of a BUFT. PAR adds this pullup to the longline on which the net is routed. The pullup is added if the net contains a single pullup and the design has been completely routed, but the net containing the pullup has one or more timing errors.

Timing-driven PAR

Timing-driven PAR is based on the Xilinx timing analysis software, an integrated static timing analysis tool that does not depend on input stimulus to the circuit. Placement and routing are executed according to timing constraints that you specify in the beginning of the design process. The timing analysis software interacts with PAR to ensure that the timing constraints imposed on your design are met.

To use timing-driven PAR, you can specify timing constraints using any of the following ways:

- Enter the timing constraints as properties in a schematic capture or HDL design entry program.
- Write your timing constraints into a User Constraints File (UCF). This file is processed by NGDBuild when the logical design database is generated.

To avoid manually entering timing constraints in a UCF file, use the Xilinx Constraints Editor, a tool that greatly simplifies constraint creation. For a detailed description of how to use the editor, see the Constraints Editor online help.

- Enter the timing constraints in the Physical Constraints File (PCF), a file that is generated by MAP. The PCF file contains any timing constraints specified using the two previously described methods and any additional constraints you enter directly in the file.

Timing-driven placement and timing-driven routing are automatically invoked if PAR finds timing constraints in the physical constraints file. The physical constraints file serves as input to the timing analysis software. The timing constraints supported by the Xilinx Development System are described in the *Constraints Guide*.

Note Depending upon the types of timing constraints specified and the values assigned to the constraints, PAR run time may be increased.

When PAR is complete, you can verify that the design's timing characteristics (relative to the physical constraints file) have been met by running TRACE (Timing Reporter and Circuit Evaluator), Xilinx's timing verification and reporting utility. TRACE, which is described in detail in the "TRACE" chapter, issues an report showing any timing warnings and errors and other information relevant to the design. There is a terse summary report of timing in the PAR report also.

Note If you are going to run a design without timing constraints, you may obtain better circuit performance by enabling the Delay Based Cleanup router pass. Alternatively, consider running timing-driven PAR by supplying timing constraints with the input design.

Automatic Timespecing

For Spartan-II and Virtex/-E designs, PAR performs automatic timespecing if it does not detect any timing constraints. This feature is only invoked if you set the –ol option to 3, 4 or 5.

When automatic timespecing is invoked, PAR analyzes your design for clock nets and attempts to increase the frequency of clocks. The alternative to automatic timespecing is manually increasing the clock frequency using a timing-driven flow, and running PAR many times with progressively higher frequencies. Each run requires an increase in the frequency specifications on clocks in your design. PAR attempts to meet these specifications, not improve them. You can continue to increase the frequency specifications until PAR can no longer meet them. At this point, your design achieves optimal clock frequency. Automatic timespecing allows you to achieve good clock frequency results in the shortest possible time.

Specifying timing constraints in your design file can still yield the best possible runtime and frequency. However, automatic timespecing provides significantly improved runtime and clock frequency compared with not using a timing-driven mode.

Note You can suppress automatic timespecing with the –x option.

Command Line Examples

Following are a few examples of PAR command lines and a description of what each does.

Example 1:

The following command places and routes the design in the file input.ncd and writes the placed and routed design to output.ncd.

```
par input.ncd output.ncd
```

Example 2:

The following command skips the placement phase and preserves all routing information without locking it (re-entrant routing). Then it runs up to 999 passes of the router or stops upon completion and conformance to timing constraints found in the pref.pcf file. Then it runs three delay-based cleanup router passes. If the design is already completely routed, the effect of this command is to just run three delay-based cleanup passes.

```
par –k –i 999 –c 0 –d 3 input.ncd output.ncd pref.pcf
```

Example 3:

The following command runs 20 place and route iterations at overall effort level 3. The mapping of the overall level (–ol) to placer effort level (–pl) and router effort level (–rl) depends on the device to which the design was mapped, and placer level and router level do not necessarily have the same value. The iterations begin at cost table entry 5. Only the best 3 output design files are saved. The output design files (in NCD format) are placed into a directory called results.dir.

```
par –n 20 –ol 3 –t 5 –s 3 input.ncd results.dir
```

Now, if you wanted to run two passes of cost-based and delay-based cleanup on the three designs saved (without running placement), you would enter this command for each design.

```
par –k –i 0 –c 2 –d 2 input.ncd output.ncd
```

Example 4:

The following command copies the input design to the output design. The placement and routing phases are skipped completely. Since a delay file is generated as a result of the command, you can use these options to check the delay times in your design without having PAR change any of the design’s placement or routing.

```
par –pr input.ncd output.ncd
```

Example 5:

The following command allows re-entrant routing. Use this command when your design is only partially routed and you want to complete it or when the design does not meet your timing constraints and additional routing passes are needed to meet the constraints. Placement and placement optimization are skipped. In this case up to 30 router passes are run (you could run up to 2000). This may result in local rip-up and reroute if 20 router passes are run with no progress.

```
par –k –i 30 input.ncd output.ncd
```

Example 6:

The following command gives you a delay report for a placed and routed file without modifying the file.

```
par –pwr input.ncd input.ncd
```

Example 7:

The following command runs PAR (using the Turns Engine) on all nodes listed in the *allnodes* file. It runs 10 place and route passes at placer effort level 3 and router effort level 2 on the mydesign.ncd file. It runs one cost-based cleanup pass of the router.

```
par –m allnodes –pl 3 –rl 2 –n 10 –i 10 –c l
mydesign.ncd output.dir
```

Guided PAR

You can use guide files to modify your design incrementally or you can integrate your design with PCI Core guide files.

Incremental Designs

Optionally, PAR reads an NCD file as a guide file to help in placing and routing the input design. This is useful if minor incremental changes have been made to create a new design. To increase productivity, you can use your last design iteration as a guide design for the next design iteration, as shown in the following figure:

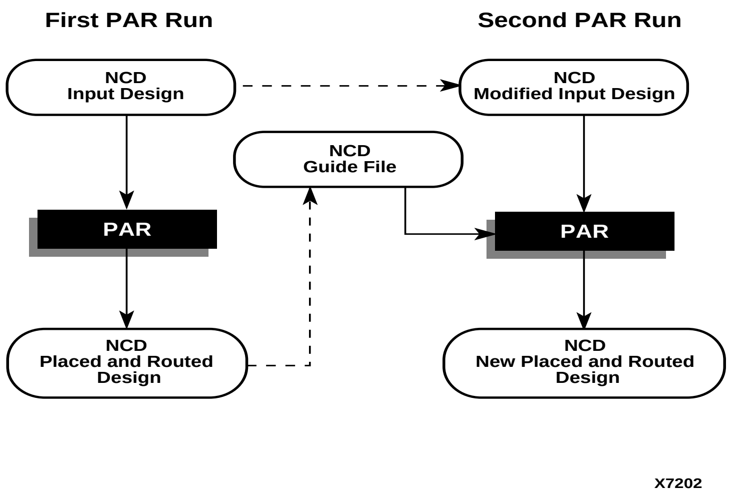

First PAR Run
Second PAR Run
NCD Input Design
NCD Modified Input Design
NCD Guide File
PAR
PAR
NCD Placed and Routed Design
NCD New Placed and Routed Design
X7202

Figure 9-4 Guided PAR for Incremental Design

Two command line options control guided PAR. The –gf option specifies the NCD guide file, and the –gm option determines whether exact or leverage mode is used to guide PAR.

The guide design is used as follows:

- If a component in the new design is constrained to the same location as a component placed in the guide file, then this component is defined as matching.

- If a component in the new design has the same name as a component in the guide design, that component matches the guide component.
- If a signal in the new design has the same name as a signal in the guide design, the signal matches the guide signal.
- Any matching component in the new design is placed in the site corresponding to the location of the matching guide component, if possible.
- Matching component’s pins are swapped to match those of the guide component with regard to matching signals, if possible.
- All of the connections between matching driver and load pins of the matching signals have the routing information preserved from the guide file, if possible.

When PAR runs using a guide design as input, PAR first places and routes any components and signals that fulfill the matching criteria described above. Then PAR places and routes the remainder of the logic.

To place and route the remainder of the logic, PAR performs the following:

- If you have selected exact guided PAR (by entering the **–gm exact** option on the PAR command line), the placement and routing of the matching logic are locked. Neither placement nor routing can be changed to accommodate the additional logic.
- If you have selected leveraged guided PAR (by entering the –**gm leverage** option on the PAR command line), PAR tries to maintain the placement and routing of the matching logic, but changes placement or routing if it is necessary in order to place and route to completion and achieve your timing constraints (if possible).

  Some cases where the leveraged mode is necessary are as follows:

  - ♦ You have added logic that makes it impossible to meet your timing constraints without changing the placement and routing in the guide design.
  - ♦ You have added logic that demands a certain site or certain routing resource, and that site or routing resource is already being used in the guide design.

For example, in XC4000EX devices BUFTs must be routed along long lines. If you add BUFTs to an XC4000EX design but your guide design uses too many of the required long lines, you can only route this design to completion if you use the leverage option.

If you enter a –gm (guide mode) option but do not specify a guide file with the –gf option, PAR is guided by the placement and routing information in the input NCD file. Depending on whether you specify exact mode or leveraged mode, PAR locks the input NCD's existing placement and routing (exact mode), or tries to maintain the placement and routing, but modifies them in an effort to place and route to completion and achieve your timing constraints (leveraged mode).

Note For Verilog or VHDL netlist designs, re-synthesizing modules typically causes signal and instance names in the resulting netlist to be significantly different from the netlist obtained in earlier synthesis runs. This occurs even if the source level Verilog or VHDL code only contains a small change. Because guided PAR depends on signal and component names, synthesis designs often have a low *match rate* when guided. Therefore, guided PAR is not recommended for most synthesis-based designs, although there may be cases where it could be a successful alternative technique.

PCI Cores

You can use a guide file to add a PCI Core, which is a standard I/O interface, to your design. The PCI Core guide file must already be placed and routed. PAR only places and routes the signals that run from the PCI Core to the input NCD design; it does not place or route any portion of the PCI Core. You can also use the resulting design (PCI Core integrated with your initial design) as a guide file. However, you must then use the **exact** option for –gm, *not* **leverage**, when generating a modified design.

Guided PAR supports precise matching of placement and routing of PCI Cores that are used as reference designs in a guide file:

- Components locked in the input design are guided by components in the reference design of a guide file in the corresponding location.

- Signals that differ only by additional loads in the input design have the corresponding pins routed according to the reference design in the guide file.
- Guide summary information in the PAR report describes the amount of logic from the reference design that matches logic in the input design.

For detailed information about designing with PCI, refer to the Xilinx PCI web page (http://www.xilinx.com/products/logicore/pci/pcilit.htm).

PAR Reports

Note Reports are formatted for viewing in a monospace (non-proportional) font. If the text editor you use for viewing the report uses a proportional font, the columns in the report do not line up correctly.

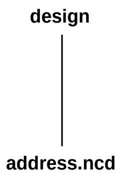
The output of PAR is a placed and routed NCD file (the output design file). In addition to the output design file, a PAR run generates a report file with a .par extension, a delay file with a .dly extension, and a pinout file with a .pad extension. The PAR file contains execution information about the place and route job as well as all constraint messages. The DLY file contains delay information about the routed nets in the design. The PAD file lists IOBs (Input/Output Blocks) on the chip and the primary pins associated with the IOBs.

If the options that you specify when running PAR are options that produce a single output design file, your output is the output design file, a PAR file, a DLY file, and a PAD file. The PAR file, the DLY file, and the PAD file all have the same root name as the output design file.

If you run multiple iterations of placement and routing, you produce an output design file, a PAR file, a DLY file, and a PAD file for each iteration. Consequently, when you run multiple iterations you have to specify a directory in which to place these files.

As the command is performed, PAR records a summary of all placement and routing iterations in one PAR file at the same level as the directory you specified, then places the output files (in NCD format) in the specified directory. Also, a PAR file, a DLY file, and a PAD file are created for each NCD file, describing in detail each individual iteration.

The following example shows a directory named design with a design file named address.ncd.



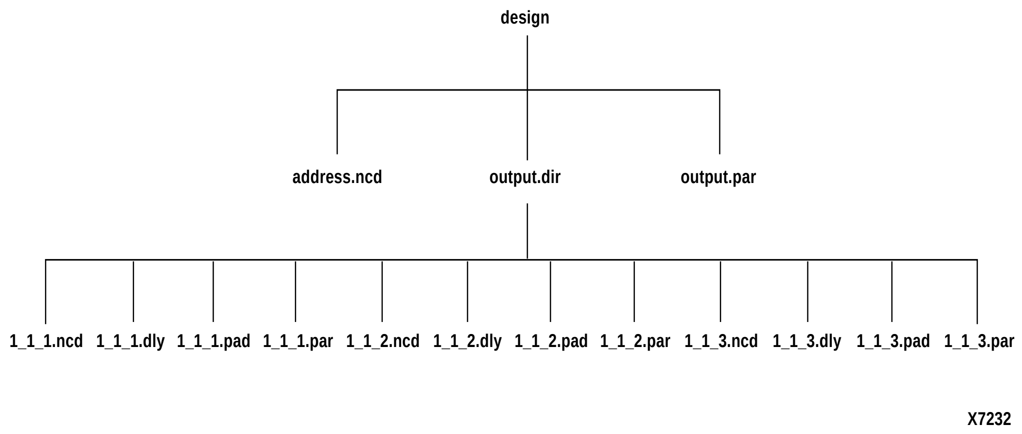
X7231

To run three iterations of place and route, using a different cost table entry each time (cost tables are explained in the "Placing" section) and specify that the resulting output be put into a directory called output.dir, use the following command:

```
par –n 3 –l 1 address.ncd output.dir
```

–n 3 is the number of iterations you want to run, **–l 1** sets the placement effort level, **address.ncd** is your input design file, and **output.dir** is the name of the directory in which you want to place the results of the PAR run.

The files resulting from the command are shown in the following figure:



The naming convention for the files, which may contain placement and routing information in varying degrees of completion, is *placer_level_router_level_cost_table.file_extension*.

In the sample above, the effort level and cost table entries start at 1 (the default value). The PAR, DLY, and PAD files are described in the following sections. When you run multiple iterations, a summary PAR report file similar to the following example is generated.

```
Release 4.1i - Par E.30
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.


Fri Jun 22 15:44:00 2001

par -ol 3 -n 5 -i 20 ndes104.mapped.ncd tougher.pcf


Level/       Design  Timing    Number       Run     NCD
Cost [ncd]   Score   Score     Unrouted     Time    Status
----------   ------  --------  --------     -----   ------------
3_3_1 *      319       0         0          02:53     Complete
3_3_2 *      319       0         0          02:54     Complete
3_3_3 *      319       0         0          02:54     Complete
3_3_4 *      328       0         0          02:54     Complete


* : Design saved.
```

The top of the summary PAR file shows the command line used to run PAR, followed by the name of any physical constraints file used.

The body of the report consists of the following columns.

Level/Cost [ncd]—indicates the effort level (1–5) at which PAR is run. In the sample above, 3_3_4 indicates placer level 3, router level 3, and the fourth cost table used.

Design Score—see "Incremental Designs" section.

Timing Score—see "Incremental Designs" section.

Number Unrouted—indicates the number of unrouted nets in the design.

Run Time—the time required to complete the job in minutes and seconds.

NCD Status—describes the state of the output NCD file generated by the PAR run. Possible values for this column are

- Complete—an NCD file has been generated by a full PAR run.
- ^C Checkpoint—initiated by the user, the PAR run was stopped at one of the PAR checkpoints. PAR produced an NCD file, but all iterations may not have been completed.

- Checkpoint—the PAR run was stopped at one of the PAR checkpoints, not because of user intervention but because of some unknown reason.
- No NCD—the PAR job was stopped prematurely and the NCD file was not checkpointed.

Intermediate Failing Timespec Summary

PAR generates an intermediate failing timespec summary only in the routing phase. The summary name is design_name.itr. The router creates this summary after an iteration not during an iteration. If an interruption occurs during an iteration (for example, with a Ctrl C), you are prompted with the following options if a time specification has failed. PAR creates an intermediate failing timespec summary generated from the end of the previous iteration. If the interrupt occurred during the first iteration, no intermediate summary is created.

1. Continue processing and ignore the interrupt.
2. Normal program exit at next check point. This will result in saving the best results so far, after concluding current processing.
3. Exit program immediately.
4. Display Failing Timespec Summary.
5. Cancel the current job and move to the next one at

If you select 3, PAR exits. If you select 4, PAR displays the contents of the ITR file on the screen and resumes execution. Option 5 allows you to terminate jobs that use the –n option for multiple iterations. If Options 4 and 5 are not applicable, a messages appears indicating that these options are not relevant.

Place and Route (PAR) Report File

The place and route (PAR) report file contains execution information about the PAR command run. The file shows the steps taken as the program converges on a placement and routing solution. A sample PAR file is shown following.

```
Release 4.1i - Par E.30
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Fri Jun 22 16:12:39 2001

par -w ndes104.mapped.ncd routed

Loading design for application par from file ndes104.mapped.ncd.
   "ndes104" is an NCD, version 2.27, device xcv100, package pq240,
speed -5
Loading  device  for  application  par  from  file  'v100.nph'  in
environment
/build/xfndry/E.30/rtf.
Device speed data version:  FINAL 1.115 2001-06-20.
Device utilization summary:

   Number of External IOBs            68 out of 166    40%
      Number of LOCed External IOBs    0 out of 68      0%

   Number of SLICEs                 1037 out of 1200   86%

   Number of GCLKs                     3 out of 4      75%
   Number of TBUFs                   146 out of 1280   11%


Overall effort level (-ol):   2 (default)
Placer effort level (-pl):    2 (default)
Placer cost table entry (-t): 1
Router effort level (-rl):    2 (default)
Extra effort level (-xe):     0 (default)

Starting initial Placement phase. REAL time: 10 secs
Finished initial Placement phase. REAL time: 11 secs
Starting the placer. REAL time: 11 secs
Placement pass 1 ........................
Placer score = 198929
Placement pass 2 ..........................................
Placer score = 192610
Placement pass 3 ...........................................
Placer score = 186808
Optimizing ...
Placer score = 152459
Placer score = 148014
Placer completed in real time: 31 secs

Dumping design to file routed.ncd.

Total REAL time to Placer completion: 33 secs
Total CPU time to Placer completion: 30 secs

0 connection(s) routed; 8947 unrouted.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 36 secs
Starting iterative routing.
Routing active signals.
...........
End of iteration 1
8947 successful; 0 unrouted; (0) REAL time: 51 secs
Constraints are met.
Total REAL time: 51 secs
Total CPU  time: 47 secs
End of route.  8947 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.

This design was run without timing constraints.  It is likely that
much better circuit performance can be obtained by trying either or
both of the following:

   - Enabling the Delay Based Cleanup router pass, if not already
enabled
  - Supplying timing constraints in the input design


Total REAL time to Router completion: 55 secs
Total CPU time to Router completion: 50 secs

Generating PAR statistics.

   The Delay Summary Report

   The Score for this design is: 310

The Number of signals not completely routed for this design is: 0
```

```
   The Average Connection Delay for this design is:       1.868 ns
   The Maximum Pin Delay is:                              9.912 ns
   The Average Connection Delay on the 10 Worst Nets is:   6.167 ns

   Listing Pin Delays by value: (ns)

d < 2.00  < d < 4.00  < d < 6.00  < d < 8.00  < d < 10.00 d>=10.00
--------  ----------  ----------  ----------  ---------- --------
  5215       3318        347            8          7          0

Dumping design to file routed.ncd.


All signals are completely routed.

Total REAL time to PAR completion: 1 mins 2 secs
Total CPU time to PAR completion: 57 secs

Placement: Completed - No errors found.
Routing: Completed - No errors found.

PAR done.
```

Summary of Report

This section includes descriptions of the various sections in the PAR report.

- In the *starting iterative routing* section, after the end of iteration 1, there is a figure in parentheses (0). This represents the timing score for the design (*not* the PAR score) at the end of the particular iteration. This figure appears in the PAR file only when timing constraints have been specified in a PCF file. When the timing score is 0 (as it is in this example after iteration 1), this means that all timing constraints have been met. This score (0) also appears at the end of the delay report section of the PAR file. The timing score at the end of the *starting iterative routing* section may not agree with the timing score in the Delay Summary Report. This can occur if a MAXSKEW constraint is scored and not met.

If the design was completely routed but failed to meet all timing constraints, the score would have been a figure other than 0. A non-zero number would appear at the end of the delay report section. This tells you immediately whether your timing constraints have been met. It is possible that the timing score shown in parentheses at the top of the file may be different from the one shown in the delay summary section of the file. The score shown in the delay summary section is always the correct one.

- The Placer score is a rating of the relative *cost* of a placement. A lower score indicates a better (less *costly*) placement.
- In the Delay Summary Report section where average delays are listed, there are two columns of figures. The first column gives the actual averages for the design. The figures in the second column, which are enclosed by parentheses, indicate the averages after the imposition of a tilde penalty.
- Timing score is always 0 (zero) if all timing constraints have been met. If not, the figure is other than 0.
- The *Score for this design* section is a rating of the routed design. The PAR file shows the total score as well as the individual factors making up the score. The score takes the following factors into account (weighted by their relative importance:
  - Number of unrouted nets (unr)
  - Number of timing constraints not met (ncst)
  - Amount (expressed in ns) that the timing constraints were not met (acst)
  - Maximum delay on a net with a weight greater than 3
  - Net weights or priorities
  - Average of all of the maximum delays on all nets (av)
  - Average of the maximum delays for the ten highest delay nets (10w)

  The lower the score, the better the result.

  The formula that produces the score is

  5000*unr + 1000*ncst + 20*acst + (delay*weight)*0.2 + av*100 + 10w*20

- The last section of the PAR file contains a summary of the delay information for the routed design. The DLY (delay) file produced by the PAR run contains more detailed timing information. The DLY file is described in the“Delay (DLY) File” section.
- If you specify a command option that produces multiple output design files, there is a PAR file indicating all of the place and route iterations performed, and individual PAR files describing placement and routing for each design file produced.

Note In PAR reporting, a tilde (~) preceding a delay value indicates that the delay value is approximate. Values with the tilde cannot be calculated exactly because of excessive delays, resistance, or capacitance on the net. You can use the PENALIZE TILDE constraint to penalize these delays by a specified percentage (see the “TRACE” chapter and the *Constraints Guide* for a description of the PENALIZE TILDE constraint).

Select I/O Utilization and Usage Summary

For the Virtex/-E/-II and Spartan-II devices, if more than one SelectI/O standard is used, an additional section on Select I/O utilization and usage summary is added to the PAR file. This section shows details for the different I/O banks. It shows the I/O standard, the output reference voltage (VCCO) for the bank, the input reference voltage (VREF) for the bank, the PAD and Pin names. In addition, the section gives a summary for each bank with the number of pads being used, the voltages of the VREFs, and the VCCOs.

For guided PAR, the PAR report displays summary information describing the total amount and percentage of components and signals in the input design guided by the reference design. The report also displays the total/percentage of components and signals from the reference design (guide file) that were used to guide the input design. See the “Guide Reporting” section.

Delay (DLY) File

The delay file is output by each PAR run and is placed in the directory with the NCD output of the design file and the PAR file. The delay file contains delay information for each net in the design and includes the following:

- A listing of the 20 nets with the longest delays. In a DLY file, maximum delays are preceded by a tilde, indicating that the delay shown is only approximate. Following each tilde delay is a figure in parentheses. This figure represents the approximate delay with a certain percentage automatically added to it (a *worst case* situation) when specified by the user in the physical constraints (PCF) file. When the Xilinx Development System's timing analysis software looks at the delays, it uses the value in parentheses rather than the approximate value represented by the tilde. For more information on the PENALIZE TILDE constraint, see the "TRACE" chapter in this manual and the *Constraints Guide.*
- A delay analysis for each net, including the net name, followed by the driver pin and the load pin or pins.

PAD File

The PAD file contains a listing of all IOBs used in the design and their associated pads. The file specifies connections to device pins (with a P prefix).

The PAD file is divided into the following sections.

- The first section lists the component name in the first column. The second column of this section lists the designations of the device pins.
- The second section lists the pin number in the first column, the component name in the second column, and any constraints assigned to the component in the third column.
- The third section lists the pinouts in the form of constraints. These constraints can be cut and pasted into a PCF file as constraints for later PAR runs.

The PAD file reports all dual-purpose pins that are used during configuration as well during normal operation.

For the Virtex/-E/-II or Spartan-II devices, when SelectI/Os are used, the PAD file also contains details of the pads that must be used for the input reference voltage (VREF), and those that must be used for the output reference voltage (VCCO). For the VREF pads, their location and the value of the input reference voltage is shown.

Guide Reporting

This report, which is included in the PAR report file, is generated with the -gf option. The report describes the criteria used to select each component and signal used to guide the design. It may also enumerate the criteria used to reject some subset of the components and signals that were eliminated as candidates.

Turns Engine (PAR Multi-Tasking Option)

This Xilinx Development System option allows you to use multiple systems (nodes) that are networked together for a multi-run PAR job, significantly reducing the total amount of time to completion. You can specify multi-tasking from the UNIX command line.

Turns Engine Overview

Before the Turns Engine was developed for the Xilinx Development System, PAR could only run multiple jobs in a linear way. The total time required to complete PAR was equal to the sum of the times that it took for each of the PAR jobs to run. This is illustrated by the following PAR command.

```
par –l 5 –n 10 –i 10 –c 1 mydesign.ncd output.dir
```

The above tells PAR to run 10 place and route passes (–**n 10**) at effort level 5 (–**l 5**), a maximum of 10 router passes (–**i 10**), and one cost-based cleanup pass (c 1). It runs each of the 10 jobs consecutively, generating an output NCD file for each job, i.e., output.dir/ 5_5_1.ncd, output.dir/5_5_2.ncd, etc. If each job takes approximately one hour, then the run takes approximately 10 hours.

The Turns Engine allows you to use all five nodes at the same time, dramatically reducing the time required for all ten jobs. To do this you must first generate a file containing a list of the node names, one per line as in the following example.

Note A pound sign (#) in the example indicates a comment.

```
# NODE names

jupiter             #Fred’s node
mars                #Harry’s node
mercury             #Betty’s node
neptune             #Pam’s node
pluto               #Mickey’s node
```

Now run the job from the command line as follows:

```
par –m nodefile_name –l 5 –n 10 –i 10 –c 1
mydesign.ncd output.dir
```

nodefile_name is the name of the node file you created.

This runs the following jobs on the nodes specified.

jupiter: par –l 5 –i 10 –c 1 mydesign.ncd output.dir/5_5_1.ncd
mars: par –l 5 –i 10 –c 1 mydesign.ncd output.dir/5_5_2.ncd
mercury: par –l 5 –i 10 –c 1 mydesign.ncd output.dir/5_5_3.ncd
neptune: par –l 5 –i 10 –c 1 mydesign.ncd output.dir/5_5_4.ncd
pluto: par –l 5 –i 10 –c 1 mydesign.ncd output.dir/5_5_5.ncd

As the jobs finish, the remaining jobs are started on the nodes until all 10 jobs are complete. Since each job takes approximately one hour, all 10 jobs complete in approximately two hours.

Note You cannot determine the relative benefits of multiple placements by running the Turns Engine with options that generate multiple placements, but do not route any of the placed designs (the –r PAR option specifies *no routing*). The design score you receive is the same for each placement. To get some indication of the quality of the placed designs, run at least one routing iteration (–i 1) on each placed design.

Turns Engine Syntax

The following is the PAR command line syntax to run the Turns Engine.

```
par –m nodelist_file –n #_of_iterations –s
#_of_iterations_to_save mapped_desgin.ncd
output_directory.dir
```

–**m** *nodelist_file* specifies the nodelist file for the Turns Engine run.

–**n** *#_of_iterations* specifies the number of place and route passes.

–s *#_of_iterations_to_save* saves only the best –s results.

*mapped design***.ncd** is the input NCD file.

*output_directory***.dir** is the directory where the best results (–s option) are saved. Files include placed and routed NCD, summary timing reports (DLY), pinout files (PAD), and log files (PAR).

Turns Engine Input Files

The following are the input files to the Turns Engine.

- NCD File—A mapped design.
- Nodelist file—A user-created ASCII file listing workstation names. The following is a sample nodelist file:

```
# This is a comment
# Note: machines are accessed by Turns Engine
# from top to bottom
# Sparc 20 machines running Solaris
kirk
spock
mccoy
krusher
janeway
picard
# Sparc 10 machines running SunOS
michael
jermaine
marlon
tito
jackie
# HPs running HP-UX
william
```

```
george
ronald
jimmy
gerald
```

Turns Engine Output Files

The naming convention for the NCD file, which may contain placement and routing information in varying degrees of completion, is placer_level_router_level_cost_table.ncd. If any of these elements are not used, they are represented by an *x*. For example, for the first design file run with the options –n 5 –t 16 –rl 4 –pl 2, the NCD output file name would be 2_4_16.ncd. The second file would be named 2_4_17.ncd. For the first design file being run with the options –n 5 –t 16 –r –pl 2, the NCD output file name would be 2_x_16.ncd. The second file would be named 2_x_17.ncd.

Homogeneous and Heterogeneous Networks

The Turns Engine can run on the following networks:

- Homogenous networks—All Solaris or all HP-UX.
- Heterogeneous networks—A mix of Solaris and HP-UX. You must have the Xilinx software and a license for each platform on which you intend to run. See the "System Requirements" section section for information on how to set up the environment variables.

Limitations

The following limitations apply to the Turns Engine.

- The Turns Engine can operate only on Xilinx FPGA families. It cannot operate on CPLDs.
- The Turns Engine can only operate on UNIX workstations.
- Each run targets the same part, and uses the same algorithms and options. Only the starting point, or the cost table entry, is varied.

System Requirements

Use the following steps to set up the system:

1. Create a file with a fixed path that can be accessed by all the systems you are using:

   /net/$ {*nodename*} /home/jim/parmsetup

2. Add the lines to set up the XILINX environment variable and the path, as shown in the following examples:

   Example for HP systems:

   ```
   export XILINX=/net/${nodename} /home/jim/xilinx
   export PATH=$XILINX/bin/hp: /usr/bin: /usr/sbin

   export SHLIB_PATH=$XILINX/bin/hp
   ```

   Example for SUN Solaris systems:

   ```
   export XILINX=/net/${nodename} /home/jim/xilinx

   export PATH=$XILINX/bin/sol: /usr/bin: /usr/sbin

   export LD_LIBRARY_PATH=$XILINX/bin/sol
   ```

   For mixed sets of systems, you need a more sophisticated script that can set up the proper environment.

3. After setting up this file, set the environment variable PAR_M_SETUPFILE to the name of your file, as shown in the following examples:

   Example for C shell:

   ```
   setenv PAR_M_SETUPFILE /net/${nodename} /home/jim/
   parmsetup
   ```

   Example for Bourne or Korn shells:

   ```
   export PAR_M_SETUPFILE=/net/${nodename} /home/jim/
   parmsetup;
   ```

Turns Engine Environment Variables

The following environment variables are interpreted by the Turns Engine manager.

- PAR_AUTOMNTPT—Specifies the network automount point. The Turns Engine uses network path names to access files. For example, a local path name to a file may be designs/cpu.ncd, but the network path name may be /home/*machine_name*/ivan/designs/cpu.ncd or /net/*machine_name*/ivan/designs/cpu.ncd. The PAR_AUTOMNT environment variable should be set to the value of the network automount point. The automount points for the examples above are /home and /net. The default value for PAR_AUTOMNT is /net.

The following command sets the automount point to /nfs. If the current working directory is /usr/*user_name*/*design_name* on node mynode, the command **cd /nfs/mynode/usr/***user_name*/*design_name* is generated before PAR runs on the machine.

```
setenv PAR_AUTOMNTPT /nfs
```

The following setting does not issue a **cd** command; you are required to enter full paths for all of the input and output file names.

```
setenv PAR_AUTOMNTPT ""
```

The following command tells the system that paths on the local workstation are the same as paths on remote workstations. This can be the case if your network does not use an automounter and all of the mounts are standardized, or if you do use an automounter and all mount points are handled generically.

```
setenv PAR_AUTOMNTPT "/"
```

- PAR_AUTOMNTTMPPT—Most networks use the /tmp_mnt temporary mount point. If your network uses a temporary mount point with a different name, like /t_mnt, then you must set the PAR_AUTOMNTTMPPT variable to the temporary mount point name. In the example above you would set PAR_AUTOMNTTMPPT to /t_mnt. The default value for PAR_AUTOMNTTMPPT is /tmp_mnt.
- PAR_M_DEBUG—Causes the Turns Engine to run in debug mode. If the Turns Engine is causing errors that are difficult to correct, you can run PAR in debug mode as follows:
    - ♦ Set the PAR_M_DEBUG variable:

      ```
      setenv PAR_M_DEBUG 1
      ```
    - ♦ Create a node list file containing only a single entry (one node). This single entry is necessary because if the node list contains multiple entries, the debug information from all of the nodes is intermixed, and troubleshooting is difficult.

- ♦ Run PAR with the –m (multi-tasking mode) option. In debug mode, all of the output from all commands generated by the PAR run is echoed to the screen. There are also additional checks performed in debug mode, and additional information supplied to aid in solving the problem.

- PAR_M_SETUPFILE—See the "System Requirements" section for a discussion of this variable.

Debugging

With the Turns Engine you may receive messages from the login process. The problems are usually related to the network or to environment variables.

- Network Problem—You may not be able to logon to the machines listed in the nodelist file.
  - ♦ Use the following command to contact the nodes:

    ```
    ping machine_name
    ```

    You should get a message that the machine is running. The ping command should also be in your path (UNIX cmd: which ping).

  - ♦ Try to logon to the nodes using the command rsh *machine_ name*. You should be able to logon to the machine. If you cannot, make sure rsh is in your path (UNIX cmd: which rsh). If rsh is in your path, but you still cannot logon, contact your network administrator.

  - ♦ Try to launch PAR on a node by entering the following command.

    ```
    rsh machine_name /bin/sh –c par.
    ```

    This is the same command that the Turns Engine uses to launch PAR. If this command is successful, everything is set up correctly for the *machine_name* node.

- Environment Problem—logon to the node with the problem by entering the following UNIX command

  ```
  rsh machine name
  ```

  Check the $XILINX, $LD_LIBRARY_PATH, and $PATH variables by entering the UNIX command **echo $** *variable_name*.

If these variables are not set correctly, check to make sure these variables are defined in your .cshrc file.

Note Some, but not all, errors in reading the .cshrc may prevent the rest of the file from being read. These errors may need to be corrected before the XILINX environment variables in the .cshrc are read. The error message /bin/sh: par not found indicates that the environment in the .cshrc file is not being correctly read by the node.

Screen Output

When PAR is running multiple jobs and is not in multi-tasking mode, output from PAR is displayed on the screen as the jobs run. When PAR is running multiple jobs in multi-tasking mode, you only see information regarding the current status of the Turns Engine. For example, when the job described in the "Turns Engine Overview" section is executed, the following screen output would be generated.

```
Starting job 5_5_1 on node jupiter
Starting job 5_5_2 on node mars
Starting job 5_5_3 on node mercury
Starting job 5_5_4 on node neptune
Starting job 5_5_5 on node pluto
```

When one of the jobs finishes, a message similar to the following is displayed.

```
Finished job 5_5_3 on node mercury
```

These messages continue until there are no jobs left to run, at which time *Finished* appears on your screen.

Note For HP workstations, you are not able to interrupt the job with Ctrl C as described following if you do not have Ctrl C set as the escape character. To set the escape character, refer to your HP manual.

You may interrupt the job at any time by pressing Ctrl C. If you interrupt the program, the following options are displayed:

1. **Continue processing and ignore the interrupt**—self-explanatory.
2. **Normal program exit at next check point**—allows the Turns Engine to wait for all jobs to finish before terminating. PAR is allowed to generate the master PAR output file (PAR), which describes the overall run results

When you select option 2, a secondary menu appears as follows:

a) **Allow jobs to finish** — current jobs finish but no other jobs start if there are any. For example, if you are running 100 jobs (–n 100) and the current jobs running are 5_5_49 and 5_5_50, when these jobs finish, job 5_5_51 is not started.

b) **Halt jobs at next checkpoint** — all current jobs stop at the next checkpoint; no new jobs are started.

c) **Halt jobs immediately** — all current jobs stop immediately; no other jobs start

3. **Exit program immediately** — all running jobs stop immediately (without waiting for running jobs to terminate) and PAR exits the Turns Engine.
4. **Add a node for running jobs** — allows you to dynamically add a node on which you can run jobs. When you make this selection, you are prompted to input the name of the node to be added to the list. After you enter the node name, a job starts immediately on that node and a *Starting job* message is displayed.
5. **Stop using a node** — allows you to remove a node from the list so that no job runs on that node.

   If you select **Stop using a node**, you must also select from the following options.

```
Which node do you wish to stop using?
   1. jupiter
   2. mars
   3. mercury
Enter number identifying the node.(<CR> to ignore)
```

Enter the number identifying the node. If you enter a legal number, you are asked to make a selection from this menu.

```
Do you wish to
  1.Terminate the current job immediately and resubmit.
  2.Allow the job to finish.
Enter number identifying choice.  (<CR> to ignore)
```

The options are described as follows:

a) **Terminate the current job immediately and resubmit**—halts the job immediately and sets it up again to be run on the next available node. The halted node is not used again unless it is enabled by the *add* function.

b) **Allow the job to finish**—finishes the node's current job, then disables the node from running additional jobs.

Note The list of nodes described above is not necessarily numbered in a linear fashion. Nodes that are disabled are not displayed. For example, if NODE2 is disabled, the next time *Stop using a node* is opted, the following is displayed.

```
Which node do you wish to stop using?
  1. jupiter
  3. mercury
Enter number identifying the node.  (<CR> to ignore)
```

6. **Display current status** — displays the current status of the Turns Engine. It shows the state of nodes and the respective jobs. Here is a sample of what you would see if you chose this option.

```
ID   NODE       STATUS        JOB      TIME

1.   jupiter    Job Running    5_5_10   02:30:45
2.   mars       Job Running    5_5_11   02:28:03
3.   mercury    Not Available
4.   neptune    Pending Term   5_5_12   02:20:01
.

.
```

A few of the entries are described as follows:

- jupiter has been running job 5_5_10 for approximately 2 1/2 hours.
- mars has been running job 5_5_11 for approximately 2 1/2 hours.
- mercury has been deactivated by the user with the *Stop using a node* option or it was not an existing node or it was not running. Nodes are *pinged* to see if they exist and are running before attempting to start a job.
- neptune has been halted *immediately* with job resubmission. The Turns Engine is waiting for the job to terminate. Once this happens the status is changed to *not available*.

There is also a *Job Finishing* status. This appears if the Turns Engine has been instructed to halt the job at the next checkpoint.

Halting PAR

Note You cannot halt PAR with Ctrl C as described following if you do not have Ctrl C set as the interrupt character. To set the interrupt character, enter **stty intr ^V^C** in the .login file or .cshrc file.

To halt a PAR operation, enter Ctrl C. In a few seconds, this message appears:

```
CNTRL-C interrupt detected.

Please choose one of the following options:
1. Continue processing and ignore the interrupt.
2. Normal program exit at next check point.
   This will result in saving the best results so
   far,
   after concluding current processing.
3. Exit program immediately.
4. Display Failing Timespec Summary.
5. Cancel the current job and move to the next one
   at
    the next check point.
Enter choice -->
```

If you have no failing time specifications or are not using the –n option, Options 4 and 5 display as follows.

```
  4. Display Failing Timespec Summary.
    (Not applicable: Data not available)
  5. Cancel the current job and move to the next one
   at
    the next check point.
    (Not applicable: Not a multi-run job.)
```

You then select one of the five options shown on the screen. The options work in this way.

- Option 1—this option causes PAR to continue operating as before the interruption. PAR then runs to completion.

- Option 2—this option continues the current place/route iteration until one of the following *check points*.
    - After constructive placement
    - After the current optimization pass
    - After the current routing iteration

    The system then exits the PAR run and saves an intermediate output file containing the results up to the check point.

    If you use this option, you may continue the PAR operation at a later time. To do this, you must look in the PAR report file to find the point at which you interrupted the PAR run. You can then run PAR on the output NCD file produced by the interrupted run, setting command line options to continue the run from the point at which it was interrupted.

    Option 2 halt during routing may be helpful if you notice that the router is performing multiple passes without improvement, and it is obvious that the router is not going to achieve 100% completion. In this case, you may want to halt the operation before it ends and use the results to that point instead of waiting for PAR to end by itself.
- Option 3—this option stops the PAR run immediately. You do not get any output file for the current place/route iteration. You do, however, still have output files for previously completed place/route iterations.
- Option 4—PAR displays the contents of the ITR file on the screen and then resumes execution.
- Option 5—Terminates current iteration if you have used the –n option and continues the next iteration.

Note If you started the PAR operation as a background process on a workstation, you must bring the process to the foreground using the fg command before you can halt the PAR operation.

After you run PAR, you can use the FPGA Editor on the NCD file to examine and edit the results. You can also perform a static timing analysis using TRACE or the Timing Analyzer. When the design is routed to your satisfaction, you can input the resulting NCD file into the Xilinx Development System's BitGen program. BitGen creates files that are used for downloading the design configuration to the target FPGA. For details on BitGen, see the “BitGen” chapter.

Chapter 10

PIN2UCF

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes PIN2UCF. The chapter contains the following sections:

- "PIN2UCF Overview"
- "PIN2UCF Syntax"
- "PIN2UCF Input Files"
- "PIN2UCF Output Files"
- "PIN2UCF Options"
- "PIN2UCF Scenarios"

PIN2UCF Overview

PIN2UCF is a program that generates pin-locking constraints in a UCF file by reading a placed NCD file for FPGAs or GYD file for CPLDs. PIN2UCF writes its output to an existing UCF file. If there is no existing UCF file, PIN2UCF creates a new file.

The following figure shows the flow through PIN2UCF.

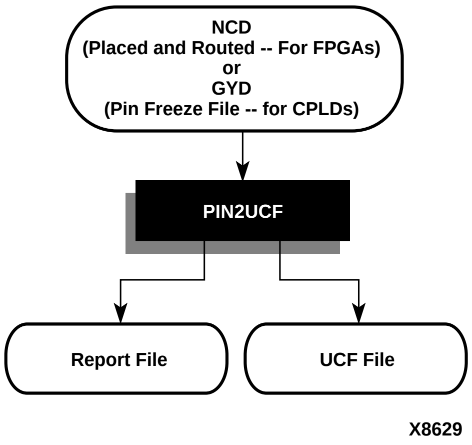

NCD
(Placed and Routed -- For FPGAs)
or
GYD
(Pin Freeze File -- for CPLDs)
PIN2UCF
Report File
UCF File
X8629

Figure 10-1 PIN2UCF Flow

The PIN2UCF is used to back-annotate pin-locking constraints to the UCF file from a successfully placed and routed design for an FPGA or a successfully fitted design for a CPLD.

The program extracts pin locations and logical pad names from an existing NCD or GYD file and writes this information to a UCF file. Pin-locking constraints are written to a PINLOCK section in the UCF file. The PINLOCK section begins with the statement #PINLOCK BEGIN and ends with the statement #PINLOCK END. By default, PIN2UCF does not write conflicting constraints to a UCF file. Prior to creating a PINLOCK section, if PIN2UCF discovers conflicting constraints, it writes information to a report file, named pinlock.rpt.

The pinlock.rpt file has the following sections:

- Constraints Conflicts Information

  This section has the following subsections. If there are no conflicting constraints, both subsections contain a single line indicating that there are no conflicts.

  - ♦ Net name conflicts on the pins
  - ♦ Pin name conflicts on the nets

  Note This section does not appear if there are fatal input errors, for example, missing inputs or invalid inputs.

- List of Errors and Warnings

  This section appears only if there are errors or warnings.

User-specified pin-locking constraints are never overwritten in a UCF file. However, if the user-specified constraints are exact matches of PIN2UCF generated constraints, a pound sign (#) is added in front of all matching user-specified location constraint statements. The pound sign indicates that a statement is a comment. To restore the original UCF file (the file without the PINLOCK section), remove the PINLOCK section and delete the pound sign from each of the user-specified statements.

Note PIN2UCF does not check if existing constraints in the UCF file are valid pin-locking constraints.

PIN2UCF writes to an existing UCF file under the following conditions:

- The contents in the PINLOCK section are all pin lock matches, and there are no conflicts between the PINLOCK section and the rest of the UCF file.
- The PINLOCK section contents are all comments and there are no conflicts outside the PINLOCK section.
- There is no PINLOCK section and no other conflicts in the UCF file.

Note Comments inside an existing PINLOCK section are never preserved by a new run of PIN2UCF. If PIN2UCF finds a CSTTRANS comment, it equates “INST *name*” to “NET name” and then checks for comments.

PIN2UCF Syntax

The following command runs PIN2UCF:

```
pin2ucf {ncd_file.ncd | pin_freeze_file.gyd} [-r
report_file_name -o output.ucf]
```

ncd_file or *pin_freeze_file* must be the name of an existing file.

PIN2UCF Input Files

PIN2UCF uses the following files as input:

- NCD file—The minimal requirement is a placed NCD file, but you would normally use a placed and routed NCD file that meets (or is fairly close to meeting) timing specifications.
- GYD file—The PIN2UCF pin-locking utility replaces the old GYD file mechanism that was used by CPLDs to lock pins. The GYD file is still available as an input guide file to control pin-locking. Running PIN2UCF is the recommended method of pin-locking to be used instead of specifying the GYD file as a guide file.

PIN2UCF Output Files

PIN2UCF creates the following files as output:

- UCF file—If there is no existing UCF file, PIN2UCF creates one. If an *output*.ucf file is not specified for PIN2UCF and a UCF file with the same root name as the design exists in the same directory as the design file, the program writes to that file automatically unless there are constraint conflicts.
- RPT file— A pinlock.rpt file is written to the current directory by default. Use the –r option to write a report file to another directory. See the "–r (Write to a Report File)" section for more information.

PIN2UCF Options

This section describes the PIN2UCF command line options.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–o (Output File Name)

```
-o outfile[.ucf]
```

By default (without the –o option), PIN2UCF writes an *ncd_file*.ucf file. The –o option specifies the name of the output UCF file for the design. You can use the –o option if the UCF file used for the design has a different root name than the design name. You can also use this option to write the pin-locking constraints to a UCF file with a different root name than the design name, or if you want to write the UCF file to a different directory.

–r (Write to a Report File)

```
-r report_file_name
```

By default (without the –r option), a pinlock.rpt file is automatically written to the current directory. The –r option writes the PIN2UCF report into the specified report file.

PIN2UCF Scenarios

The following table lists the PIN2UCF scenarios.

Table 10-1 PIN2UCF Scenarios

Scenario	PIN2UCF Behavior	Files Created or Updated
No UCF file is present.	PIN2UCF creates a UCF file and writes the pin-locking constraints to the UCF file.	pinlock.rpt *design_name*.ucf
UCF file is present. There are no pin-locking constraints in the UCF file, or this file contains some user-specified pin-locking constraints outside of the PINLOCK section. None of the user-specified constraints conflict with the PIN2UCF generated constraints.	PIN2UCF appends the pin-locking constraints in the PINLOCK section to the end of the file.	pinlock.rpt *design_name*.ucf

Table 10-1 PIN2UCF Scenarios

Scenario	**PIN2UCF Behavior**	**Files Created or Updated**
UCF file is present. The UCF file contains some user-specified pin-locking constraints either inside or outside of the PINLOCK section. Some of the user-specified constraints conflict with the PIN2UCF generated constraints	PIN2UCF does not write the PINLOCK section. Instead, it exits after providing an error message. It writes a list of conflicting constraints.	pinlock.rpt
UCF file is present. There are no pin-locking constraints in the UCF file. There is a PINLOCK section in the UCF file generated from a previous run of PIN2UCF or manually created by the user. None of the constraints in the PINLOCK section conflict with PIN2UCF generated constraints.	PIN2UCF writes a new PINLOCK section in the UCF file after deleting the existing PINLOCK section. The contents of the existing PINLOCK section are moved to the new PINLOCK section.	pinlock.rpt *design_name*.ucf

Chapter 11

TRACE

This program is compatible with the following families:

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA

This chapter describes TRACE. The chapter contains the following sections:

TRACE Overview

TRACE (Timing Reporter And Circuit Evaluator) provides static timing analysis of a design based on input timing constraints.

Note On the command line, the TRACE command is entered as **trce** (without an “A”).

TRACE performs two major functions.

- Timing verification—the process of verifying that the design meets your timing constraints.

- Reporting—the process of enumerating input constraint violations and placing them into an accessible file. TRACE can be run on unplaced designs, completely placed and routed designs, or designs that are placed and routed to any degree of completion.

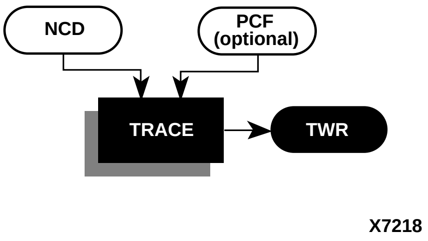


Figure 11-1 TRACE

TRACE Syntax

Use the following syntax to run TRACE:

```
trce [options] design[.ncd] [constraint[.pcf]]
```

options can be any number of the TRACE options listed in the "TRACE Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

design is the name of the input physical design file. If you enter a file name with no extension, TRACE looks for an NCD file with the name you specified.

constraint specifies the name of a timing physical constraints file. This file is used to define timing constraints for the design. If you do not specify a physical constraints file, TRACE looks for one with the same root name as the NCD file.

TRACE Input Files

Input to TRACE is a mapped NCD design and an optional physical constraints (PCF) file based upon timing constraints that you specify. Constraints can indicate such things as clock speed for input signals, the external timing relationship between two or more signals, absolute maximum delay on a design path, or a general timing requirement for a class of pins.

Input files to TRACE are as follows:

- NCD file—a mapped design. The type of timing information you receive depends on whether the design is unplaced, placed only, or placed and routed.
- PCF file—an optional user-modifiable ASCII Physical Constraints File produced by MAP. The PCF file contains timing constraints used in the TRACE timing analysis.

  Note The Innoveda CAE tools create a file with a .pcf extension when generating a plot of an Innoveda schematic. This PCF file is not related to a Xilinx PCF file. Because TRACE automatically reads a PCF file with the same root name as your design file, make sure your directory does not contain an Innoveda PCF file with the same root name as your NCD file.

TRACE Output Files

Output from TRACE is a formatted timing report (TWR) file. There are three different types of timing reports: summary report, error report and verbose report. The type of report produced is determined by the TRACE command line options you enter, as shown in the following table.

Table 11-1 TRACE Options and Reports

TRACE Option	Timing Report Produced
No –e or –v	Summary report
–e	Error report
–v	Verbose report

Optionally, you can use the –xml option (see the "–xml (XML Output File Name)" section) to generate an XML timing report (TWX) file. You can view this report with the Timing Analyzer. The –e and –v options apply to the TWX file as well as the TWR file.

Note In addition to the timing (TWR) report, you can specify –tsi on the command line to generate a Timespec Interaction Report (TSI). See the"TSI Report" section in this chapter for details.

TRACE Options

This section describes the options to the TRACE command.

–a (Advanced Analysis)

The –a option can only be used if you are not supplying any timing constraints (in a PCF file) to TRACE. The –a option writes out a timing report with the following information:

- An analysis that enumerates all clocks and the required OFFSETs for each clock.
- An analysis of paths having only combinatorial logic, ordered by delay.

This information is supplied in place of the default information for the output timing report type (summary, error, or verbose).

Note An analysis of the paths associated with a particular clock signal includes a hold violation (race condition) check only for paths whose start and endpoints are registered on the same clock edge.

–e (Generate an Error Report)

```
-e [limit]
```

The –e option causes the timing report to be an error report instead of a summary report. See the “Error Report” section for a sample error report.

The report has the same root name as the input design and a .twr extension. You can assign a different root name for the report on the command line with the –o option, but the extension must be .twr.

The optional *limit* is an integer limit on the number of items reported for each timing constraint in the report file. The value of *limit* must be an integer from 0 to 32,000 inclusive. The default is 3.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the “–f (Execute Commands File)” section of the “Introduction” chapter.

–l (Limit Timing Report)

```
–l limit
```

The –l option is used to limit the number of items reported for each timing constraint in the report file. The value of *limit* must be an integer from 0 to 32,000 inclusive.

Note The higher the limit value, the longer it takes to generate the timing report.

–o (Output File Name)

```
–o outfile[.twr]
```

The –o option specifies the name of the output timing report. The .twr extension is optional. If –o is not used, the output timing report will have the same root name as the input design (NCD) file.

–s (Change Speed)

```
–s [speed]
```

The –s option overrides the device speed contained in the input NCD file and instead performs an analysis for the device speed you specify. The –s option applies to whichever report type you produce in this TRACE run. The option allows you to see if faster or slower speed grades meet your timing requirements.

The device *speed* can be entered with or without the leading dash. For example, both **–s 3** and **–s –3** are valid entries.

Some architectures support minimum timing analysis. The command line syntax for min timing analysis is: trace –s min. Do not place a leading dash before min.

Note The –s option only changes the speed grade for which the timing analysis is performed; it does not save the new speed grade to the NCD file.

–skew (Analyze Clock Skew for All Clocks)

This –skew option analyzes clock skew for all clocks including those using non-dedicated clock routing resources.

–stamp (Generates STAMP timing model files)

–stamp *stampfile design*.**ncd**

Note The *stampfile* entry must precede the NCD file entry on the command line.

When you specify the –stamp option, TRACE generates a pair of STAMP timing model files, stampfile.mod and stampfile.data, that characterize the design's timing.

The STAMP compiler can be used for any board when performing static timing analysis.

There are four methods of running TRACE with the STAMP option to obtain a complete STAMP model report.

- Run with advanced analysis using the –a option.
- Run using default analysis (with no constraint file and without advanced analysis).
- Construct constraints to cover all paths in the design.
- Run using the unconstrained path report (–u option) for constraints which only partially cover the design.

For either of the last two options, do not include controls or TIGs, or be aware that those paths are not part of the model.

–tsi (Generate a Timespec Interaction Report)

–tsi *designfile*.**tsi** *designfile*.**ncd** *designfile*.**pcf**

When you specify the –tsi option, TRACE generates a Timespec Interaction Report. You can specify any name for the .tsi file. The file name is independent of the NCD and PCF file names. You can also specify the NCD and PCF files from which the Timespec Interaction Report analyzes constraints.

–u (Report Uncovered Paths)

–u [*limit:0,32000*]

The –u option reports delays for paths that are not covered by timing constraints. The option adds an *unconstrained path analysis* constraint to your existing constraints. This constraint performs a default path enumeration on any paths for which no other constraints apply. The default path enumeration includes circuit paths to data and clock pins on sequential components and data pins on primary outputs.

The limit variable is used to limit the number of unconstrained paths reported for each timing constraint in the report file. The value of *limit* must be an integer from 0 to 32,000 inclusive.

In the TRACE report, the following information is included for the unconstrained path analysis constraint.

- The minimum period for all of the uncovered paths to sequential components.
- The maximum delay for all of the uncovered paths containing only combinatorial logic.
- For a verbose report only, a listing of periods for sequential paths and delays for combinatorial paths. The list is ordered by delay in descending order, and the number of entries in the list can be controlled by specifying a limit when you enter the –v (Generate a Verbose Report) command line option.

Note Register-to-register paths included in the unconstrained path report will undergo a hold violation (race condition) check only for paths whose start and endpoints are registered on the same clock edge.

–v (Generate a Verbose Report)

–v [*limit:0,32000*]

The –v option generates a verbose report. The report has the same root name as the input design and a .twr. You can assign a different root name for the report on the command line, but the extension must be .twr.

The limit variable is used to limit the number of items reported for each timing constraint in the report file. The value of *limit* must be an integer from 0 to 32,000 inclusive.

–xml (XML Output File Name)

The –xml option specifies the name of the XML output timing report (TWX) file. The .twx extension is optional.

```
–xml outfile[.twx]
```

Note The XML report is not formatted and can only be viewed with the Timing Analyzer.

TRACE Command Line Examples

The following command verifies the timing characteristics of the design named design1.ncd, generating a summary timing report. Timing constraints contained in the file group1.pcf are the timing constraints for the design. This generates the report file design1.twr.

```
trce design1.ncd group1.pcf
```

The following command produces a file listing all delay characteristics for the design named design1.ncd, using the timing constraints contained in the file group1.pcf. The verbose report file is called output.twr.

```
trce –v design1.ncd group1.pcf –o output.twr
```

The following command produces a file listing all delay characteristics for the design named design1.ncd, using the timing constraints contained in the file group1.pcf. The verbose report file is called design1.twr, and the verbose XML report file is called output.twx.

```
trce –v design1.ncd group1.pcf –xml output.twx
```

The following command analyzes the file design1.ncd and reports on the three worst errors for each constraint in timing.pcf. The report is called design1.twr.

```
trce   –e 3   design1.ncd   timing.pcf
```

The following command generates a TSI report in addition to a summary timing report. The TSI report is called design1.tsi (specified on the command line). The summary timing report is called design1.twr.

```
trce –tsi design1.tsi design1.ncd timing.pcf
```

TRACE Reports

TRACE output is a formatted ASCII timing report file that provides information on how well the timing constraints for the design have been met. The file is written into your current working directory and has a .twr extension. The default name for the file is the same root name as the NCD file. You can designate a different root name for the file, but it must have a .twr extension. The extension .twr is assumed if not specified.

The timing report lists statistics on the design, any detected timing errors, and a number of warning conditions.

Timing errors indicate absolute or relative timing constraint violations, and include the following:

- Path delay errors—where the path delay exceeds the maximum delay constraint for a path.
- Net delay errors—where a net connection delay exceeds the maximum delay constraint for the net.
- Offset errors—where either the delay offset between an external clock and its associated data-in pin is insufficient to meet the internal logic's timing requirements or the delay offset between an external clock and its associated data-out pin exceeds the external logic's timing requirements.
- Net skew errors—where skew between net connections exceeds the maximum skew constraint for the net.

Timing errors may require design modifications, running PAR, or both.

Warnings point out potential problems such as circuit cycles or a constraint that does not define any paths.

Three types of reports are available. You determine the report type by entering the appropriate option entry on the workstation or PC command line or by selecting the type of report from the Timing Analyzer (see the "TRACE Options" section). Each type of report is described in the "Reporting with TRACE" section.

In addition to the formatted ASCII timing report (TWR) file, you can generate an XML timing report (TWX) file with the –xml option. The XML report is not formatted and can only be viewed with the Timing Analyzer.

Timing Verification with TRACE

TRACE checks the delays in the NCD design file against your timing constraints. If delays are exceeded, TRACE issues the appropriate timing error.

Net Delay Constraints

The delay for a constrained net is checked to ensure that the routedelay is less than or equal to the netdelayconstraint.

`routedelay ≤ netdelayconstraint`

routedelay is the signal delay between the driver pin and the load pin(s) on a net. This is an estimated delay if the design is placed but not routed.

Any nets showing delays that do not meet this condition generate timing errors in the timing report.

Net Skew Constraints

Signal skew on a net with multiple load pins is the difference between minimum and maximum load delays.

`signalskew = (maxdelay - mindelay)`

maxdelay is the maximum delay between the driver pin and a load pin.

mindelay is the minimum delay between the driver pin and a load pin.

Note Register-to-register paths included within a MAXDELAY constraint report will undergo a hold violation (race condition) check only for paths whose start and endpoints are registered on the same clock edge.

For constrained nets in the PCF, skew is checked to ensure that the signalskew is less than or equal to the maxskewconstraint.

`signalskew ≤ maxskewconstraint`

If the skew is found to exceed the maximum skew constraint, the timing report shows a skew error.

Path Delay Constraints

The pathdelay equals the sum of logic (component) delay, route (wire) delay, and setup time (if any), minus clock skew (if any).

```
pathdelay = logicdelay + routedelay + setuptime -
clockskew
```

The delay for constrained paths is checked to ensure that the pathdelay is less than or equal to the maxpathdelayconstraint.

pathdelay ≤ maxpathdelayconstraint.

logicdelay is the pin-to-pin delay through a component.

routedelay is the signal delay between component pins in a path. This is an estimated delay if the design is placed but not routed.

setuptime (for clocked paths only) is the time that data must be present on an input pin before the arrival of the triggering edge of a clock signal.

clockskew (for register-to-register clocked paths only) is the difference between the amount of time the clock signal takes to reach the destination register and the amount of time the clock signal takes to reach the source register. Clock skew is discussed in the following section.

Paths showing delays that do not meet this condition generate timing errors in the timing report.

Clock Skew and Setup Checking

Clock skew must be accounted for in register-to-register setup checks. For register-to-register paths, the data delay must reach the destination register within a single clock period for the destination register. The timing analysis software ensures that any clock skew between the source and destination registers is accounted for in this check.

Note In default mode, that is, without using the –skew option, only dedicated clock resource skew accounting is performed. With the –skew option, non-dedicated clock skew accounting is also performed.

A setup check performed on register-to-register paths checks the following condition:

```
Slack = constraint + Tsk - (Tpath + Tsu)
```

constraint is the required time interval for the path, either specified explicitly by you with a FROM TO constraint, or derived from a PERIOD constraint.

Tpath is the summation of component and connection delays along the path (including the Tcko delay from the source register).

Tsu (setup) is the setup requirement for the destination register.**Tsk (skew)** is the difference between the arrival time for the destination register and the source register.

Negative slack indicates that a setup error may occur, because the data from the source register does not set up at the target register for a subsequent clock edge.

In the following figure, the clock skew Tsk is the delay from the clock input (CLKIOB) to register D (TclkD) less the delay from the clock input (CLKIOB) to register S (TclkS). Negative skew relative to the destination reduces the amount of time available for the data path, and positive skew relative to the destination register is truncated to zero.

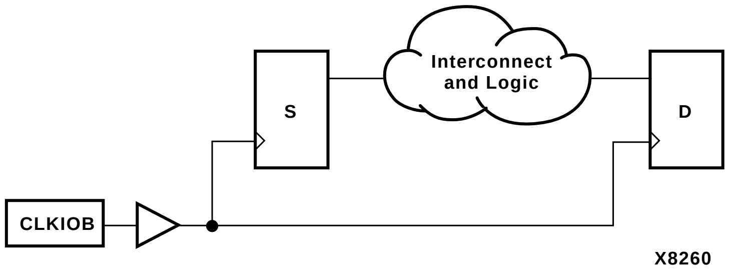


Figure 11-2 Clock Skew Example

Because the total clock path delay is used to determine the clock arrival times at the source register (TclkS) and the destination register (TclkD), this check still applies if the source and destination clocks originate at the same chip input but travel through different clock buffers and/or routing resources, as shown in the following figure.

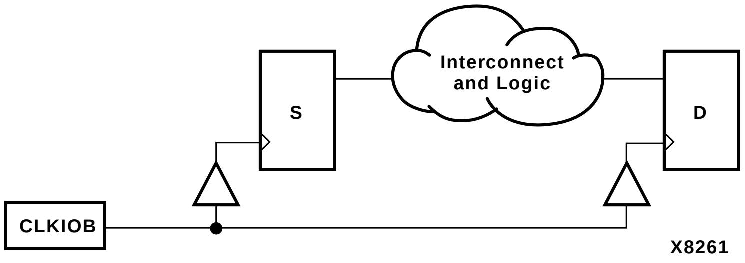


Figure 11-3 Clock Passing Through Multiple Buffers

When the source and destination clocks originate at different chip inputs, no obvious relationship between the two clock inputs exists for the timing software (because the software cannot determine the clock arrival time or phase information).

For FROM TO specifications, the software assumes you have taken into account the external timing relationship between the chip inputs. The software assumes both clock inputs arrive simultaneously, and the difference between the destination clock arrival time (TclkD) and the source clock arrival time (TclkS) does not account for any difference in the arrival times at the two chip clock inputs.

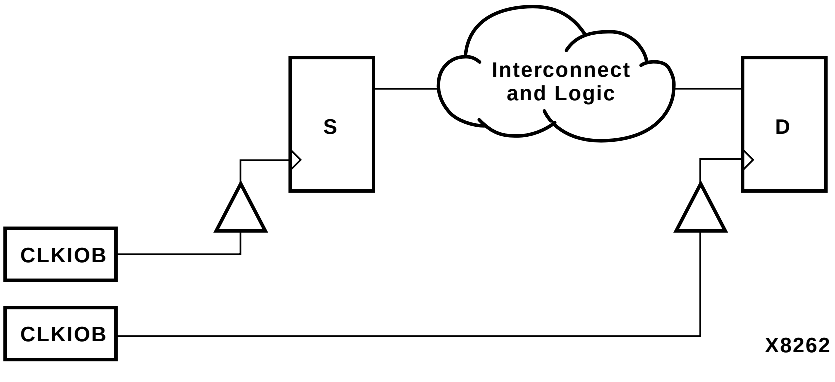


Figure 11-4 Clocks Originating at Different Device Inputs

The clock skew Tsk is not accounted for in setup checks covered by PERIOD constraints where the clock paths to the source and destination registers originate at different clock inputs.

Reporting with TRACE

The timing report produced by TRACE is a formatted ASCII (TWR) file prepared for a particular design. It reports statistics on the design, a summary of timing warnings and errors, and optional detailed net and path delay reports.

In addition to the TWR file, you can generate an XML timing report (TWX) file using the –xml option. The contents of the XML timing report are identical to the ASCII timing report. The XML report is not formatted and can only be viewed with the Timing Analyzer.

Note The ASCII TRACE reports are formatted for viewing in a monospace (non-proportional) font. If the text editor you use for viewing the reports uses a proportional font, the columns in the reports do not line up correctly.

This section describes the following types of timing reports generated by TRACE.

- The summary report—Contains summary information, design statistics, and statistics for each constraint in the PCF file.
- The error report—Lists timing errors and associated net/path delay information.
- The verbose report—Lists delay information for all nets and paths.

In each type of report, the header specifies the command line used to generate the report, the type of report, the input design name, the optional input physical constraints file name, and device and speed data for the input NCD file. At the end of each report is a timing summary, which includes the following information:

- The number of timing errors found in the design. This information appears in all reports
- A timing score, showing the total amount of error (in picoseconds) for all timing constraints in the design
- The number of paths and nets covered by the constraints
- The number of route delays and the percentage of connections covered by timing constraints

Note The percentage of connections covered by timing constraints is given in a “% coverage” statistic. The statistic does not indicate the percentage of paths covered; it indicates the percentage of connections covered. Even if you have entered constraints that cover all paths in the design, this percentage may be less than 100%, since some connections are never included for static timing analysis (for example, connections to the STARTUP component).

In the following sections, a description of each report is accompanied by a sample.

The following is a list of additional information on timing reports:

- For any of the three types of reports, if you specify a physical constraints file that contains invalid data, a list of physical constraints file errors appears at the beginning of the report. These include errors in constraint syntax.
- In a timing report, a tilde (~) preceding a delay value indicates that the delay value is approximate. Values with the tilde cannot be calculated exactly because of excessive delays, resistance, or capacitance on the net, that is, the path is too complex to calculate accurately.

  The tilde (~) also means that the path may exceed the numerical value listed next to the tilde by as much as 20%. You can use the PENALIZE TILDE constraint to penalize these delays by a specified percentage (see the *Constraints Guide* for a description of the PENALIZE TILDE constraint).
- In a timing report, an “e” preceding a delay value indicates that the delay value is estimated because the path is not routed.
- TRACE detects when a path cycles (that is, when the path passes through a driving output more than once), and reports the total number of cycles detected in the design. When TRACE detects a cycle, it disables the cycle from being analyzed. If the cycle itself is made up of many possible routes, each route is disabled for all paths which converge through the cycle in question and the total number is included in the reported cycle tally.

  A path is considered to cycle outside of the influence of other paths in the design. Thus if a valid path follows a cycle from another path, but actually converges at an input and not a driving output, the path is not disabled and will contain the elements of the cycle which may be disabled on another path.

- In Xilinx FPGAs, 3-state buffer (BUFT) outputs are always routed on longlines. Pull-up resistors may also be tied to these longlines. The timing effects of a BUFT / pull-up combination is handled differently in the various FPGA architectures.

  Note If you are using the BUFT as a MUX, do not use a pull-up.

  - In 4000E / L and Spartan designs, the delay associated with the longline is built into the component delay for the BUFT, and is not included in the delay reported for the net on the longline.
  - In XC4000EX / XL / XV designs, the net delay on the longline is computed and reported as if the pull-up (and not the BUFT output) is driving the net. If you want the delay to be computed with the BUFT driving the net, do not include any pull-ups at the output of the BUFT.

- Error counts reflect the number of path endpoints (register setup inputs, output pads) that fail to meet timing specifications, not the number of paths that fail the specification, as shown in the following figure.

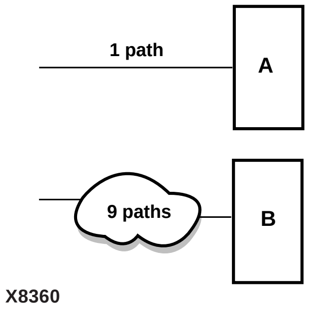


Figure 11-5 Error Reporting

If an error is generated at both the endpoints of A and B, the timing report would list two errors—one for each endpoint.

- In Virtex-II designs, the MAP program places information that is necessary to identify dedicated clocks in the PCF file. You must use a PCF generated by the MAP program to ensure accurate timing analysis on these designs.

Data Sheet Reports

The summary, error, and verbose reports contain a data sheet report. This report only includes I/Os that are covered by the specified physical timing constraints, if any. A warning is issued if the report does not cover any I/Os of the design due to the specified timing constraints. In the absence of a physical constraint file, all I/O timing is analyzed and reported (less the effects of any default path tracing controls). Unconstrained path analysis can be used with a constraint file to increase the coverage of the report to include paths not explicitly specified in the constraints file. The report includes the source and destination PAD names, and either the propagation delay between the source and destination or the setup and hold requirements for the source relative to the destination. This report summarizes the following delay characteristics for the design.

- External setup/hold requirements

  The maximum setup and hold times of device data inputs are listed relative to each clock input. When two or more paths from a data input exist relative to a device clock input, the worst-case setup and hold times are reported. One worst-case setup and hold time is reported for each data input and clock input combination in the design.

  Following is an example of an external setup/hold requirement in the data sheet report:

  ```
  Setup/Hold to clock ck1_i

  ---------------+------------+------------+

                 | Setup to   |Hold to     |

  Source Pad     |clk (edge) |clk (edge)|

  ---------------+------------+------------+

  start_i        |2.816(R)    |0.000(R)    |

  ---------------+------------+------------+
  ```

- Clock-to-output propagation delays
- The maximum propagation delay from clock inputs to device data outputs are listed for each clock input. When two or more paths from a clock input to a data output exist, the worst-case

propagation delay is reported. One worst-case propagation delay is reported for each data output and clock input combination.

Following are two examples of clock-to-output propagation delays in the data sheet report:

```
Clock ck1_i to Pad

---------------+-----------+

               |clk (edge)|

Destination Pad|to PAD    |

---------------+-----------+

out1_o         |   16.691(R)|

---------------+-----------+

Clock to Setup on destination clock ck2_i

---------------+---------+---------+---------+--

|Src/Dest| Src/Dest| Src/Dest| Src/Dest|

Source Clock|Rise/Rise|Fall/Rise|Rise/Fall|Fall/

Fall|

---------------+---------+---------+---------+-+

ck2_i        | 12.647  |       |       |     |

ck1_i        |10.241   |       |       |     |

---------------+---------+---------+---------+-+
```

- Input-to-output propagation delays

  The maximum propagation delay from each device input to each device output is reported if a combinational path exists between the device input and output. When two or more paths exist between a device input and output, the worst-case propagation delay is reported. One worst-case propagation delay is reported for every input and output combination in the design.

Following are examples of input-to-output propagation delays:

```
Pad to Pad

-----------------------------------------------

Source Pad   |Destination Pad|Delay  |

--------------+-------------+---------+

BSLOT0       |D0S           |37.534   |

BSLOT1       |D09           |37.876   |

BSLOT2       |D10           |34.627   |

BSLOT3       |D11           |37.214   |

CRESETN      |VCASN0        |51.846   |

CRESETN      |VCASN1        |51.846   |

CRESETN      |VCASN2        |49.776   |

CRESETN      |VCASN3        |52.408   |

CRESETN      |VCASN4        |52.314   |

CRESETN      |VCASN5        |52.314   |

CRESETN      |VCASN6        |51.357   |

CRESETN      |VCASN7        |52.527   |

--------------+-------------+---------
```

There are four methods of running TRACE to obtain a complete data sheet report.

- Run with advanced analysis (–a)
- Run using default analysis (that is, with no constraint file and without advanced analysis)
- Construct constraints to cover all paths in the design
- Run using the unconstrained path report for constraints which only partially cover the design

For either of the last two options, you should not have any path controls or TIGs or be aware that those paths will not be part of the report.

Guaranteed Setup and Hold Reporting

Guaranteed setup and hold values obtained from speed files are used in the data sheet reports for IOB input registers when these registers are clocked by specific clock routing resources and when the guaranteed setup and hold times are available for a specified device and speed.

Specific clock routing resources are clock networks that originate at a clock IOB, use a clock buffer to reach a clock routing resource and route directly to IOB registers.

Guaranteed setup and hold times are also used for reporting of input OFFSET constraints.

The following figure and text describes the external setup and hold time relationships.

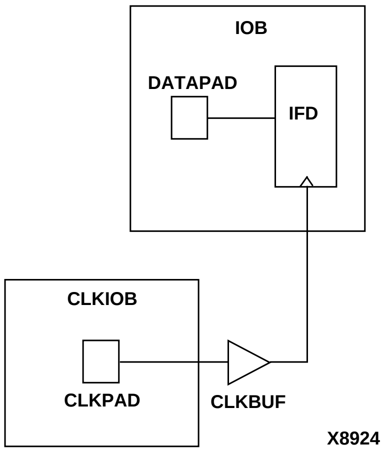


Figure 11-6 Guaranteed Setup and Hold

The pad CLKPAD of clock input component CLKIOB drives a global clock buffer CLKBUF, which in turn drives an input flip-flop IFD. The input flip-flop IFD clocks a data input driven from DATAPAD within the component IOB.

Setup Times

The external setup time is defined as the setup time of DATAPAD within IOB relative to CLKPAD within CLKIOB. When a guaranteed external setup time exists in the speed files for a particular DATAPAD and the CLKPAD pair and configuration, this number is used in timing reports. When no guaranteed external setup time exists in the speed files for a particular DATAPAD and CLKPAD pair, the external setup time is reported as the maximum path delay from DATAPAD to the IFD plus the maximum IFD setup time, less the minimum of maximum path delay(s) from the CLKPAD to the IFD.

Hold Times

The external hold time is defined as the hold time of DATAPAD within IOB relative to CLKPAD within CLKIOB. When a guaranteed external hold time exists in the speed files for a particular DATAPAD and the CLKPAD pair and configuration, this number is used in timing reports.

When no guaranteed external hold time exists in the speed files for a particular DATAPAD and CLKPAD pair, the external hold time is reported as the maximum path delay from CLKPAD to the IFD plus the maximum IFD hold time, less the minimum of maximum path delay(s) from the DATAPAD to the IFD.

Summary Report

The summary report includes the name of the design file being analyzed, the device speed and report level, followed by a statistical brief that includes the summary information and design statistics. The report also list statistics for each constraint in the PCF file, including the number of timing errors for each constraint.

A summary report is produced when you do not enter an –e (error report) or –v (verbose report) option on the TRACE command line.

Two sample summary reports are shown below. The first sample shows the results without having a physical constraints file. The second sample shows the results when a physical constraints file is specified.

If no physical constraints file exists or if there are no timing constraints in the PCF file, TRACE performs default path and net enumeration to provide timing analysis statistics. Default path enumeration includes all circuit paths to data and clock pins on sequential components and all data pins on primary outputs. Default net enumeration includes all nets.

Summary Report (Without a Physical Constraints File Specified)

The following sample summary report represents the output of this TRACE command.

```
trce –o summary.twr ramb16_s1.ncd
```

The name of the report is summary.twr. No preference file is specified on the command line, and the directory containing the file ram16_s1.ncd did not contain a PCF file called ramb16_s1.pcf.

```
------------------------------------------------------------------
Xilinx TRACE
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.
Design file:              ramb16_s1.ncd
Device,speed:             xc2v250,-6
Report level:             summary report
------------------------------------------------------------------

WARNING:Timing  -  No  timing  constraints  found,  doing  default
enumeration.
Asterisk (*) preceding a constraint indicates it was not met.
------------------------------------------------------------------
  Constraint                       | Requested  | Actual    | Logic

                                   |            |           | Levels
------------------------------------------------------------------
  Default period analysis          |            | 2.840ns    | 2
------------------------------------------------------------------
  Default net enumeration          |            | 0.001ns    |
------------------------------------------------------------------

All constraints were met.

Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)
```

```
Setup/Hold to clock clk
---------------+-----------+-----------+
               |  Setup to  |  Hold to   |
Source Pad     | clk (edge) | clk (edge) |
---------------+-----------+-----------+
ad0            |    0.263(R)|    0.555(R)|
ad1            |    0.263(R)|    0.555(R)|
ad10           |    0.263(R)|    0.555(R)|
ad11           |    0.263(R)|    0.555(R)|
ad12           |    0.263(R)|    0.555(R)|
ad13           |    0.263(R)|    0.555(R)|
.
.
.
---------------+-----------+-----------+
Clock clk to Pad
---------------+-----------+
               | clk (edge) |
Destination Pad|   to PAD   |
---------------+-----------+
d0             |    7.496(R)|
---------------+-----------+

Timing summary:
---------------
Timing errors: 0  Score: 0

Constraints cover 20 paths, 21 nets, and 21 connections (100.0%
coverage)

Design statistics:
   Minimum period:   2.840ns (Maximum frequency: 352.113MHz)
   Maximum combinational path delay:   6.063ns
   Maximum net delay:   0.001ns
Analysis completed Wed Mar  8 14:52:30 2000
------------------------------------------------------------------
```

Summary Report (With a Physical Constraints File Specified)

The following sample summary report represents the output of this TRACE command:

```
trce –o summary1.twr ramb16_s1.ncd clkperiod.pcf
```

The name of the report is summary1.twr. The timing analysis represented in the file were performed by referring to the constraints in the file clkperiod.pcf.

```
------------------------------------------------------------------
Xilinx TRACE
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.

Design file:              ramb16_s1.ncd
Physical constraint file: clkperiod.pcf
Device,speed:             xc2v250,-6
Report level:             summary report
------------------------------------------------------------------

Asterisk (*) preceding a constraint indicates it was not met.

------------------------------------------------------------------
Constraint                        | Requested | Actual    | Logic
                                  |           |           | Levels
------------------------------------------------------------------
TS01 = PERIOD TIMEGRP "clk" 10.0ns |          |         |
------------------------------------------------------------------
OFFSET = IN 3.0 ns AFTER COMP "clk" TIMEG | 3.000ns| 8.593ns    2
RP "rams"
------------------------------------------------------------------
* TS02 = MAXDELAY FROM TIMEGRP "rams" TO TI | 6.000ns  | 6.063ns   |2
  MEGRP "pads" 6.0 nS                       |          |           |
------------------------------------------------------------------

1 constraint not met.

Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)
```

```
Setup/Hold to clock clk
---------------+-----------+-----------+
               |  Setup to  |  Hold to   |
Source Pad     | clk (edge) | clk (edge) |
---------------+-----------+-----------+
ad0            |    0.263(R)|    0.555(R)|
ad1            |    0.263(R)|    0.555(R)|
ad10           |    0.263(R)|    0.555(R)|
ad11           |    0.263(R)|    0.555(R)|
ad12           |    0.263(R)|    0.555(R)|
ad13           |    0.263(R)|    0.555(R)|
.
.
.
---------------+-----------+-----------+
Clock clk to Pad
---------------+-----------+
               | clk (edge) |
Destination Pad|   to PAD   |
---------------+-----------+
d0             |    7.496(R)|
---------------+-----------+

Timing summary:
---------------

Timing errors: 1  Score: 63

Constraints cover 19 paths, 0 nets, and 21 connections (100.0%
coverage)

Design statistics:
   Maximum path delay from/to any node:   6.063ns
   Maximum input arrival time after clock:   8.593ns

Analysis completed Wed Mar  8 14:54:31 2000
------------------------------------------------------------------
```

When the physical constraints file includes timing constraints, the summary report lists the percentage of all design connections covered by timing constraints. If there are no timing constraints, the report shows 100 percent coverage. An asterisk precedes constraints that fail.

Error Report

The error report lists timing errors and associated net / path delay information. Errors are ordered by constraint and, within constraints, by slack (the difference between the constraint and the analyzed value, with a negative slack indicating an error condition). The maximum number of errors listed for each constraint is set by the limit you enter on the command line. The error report also contains a list of all time groups defined in the PCF file and all of the members defined within each group.

The main body of the error report lists all timing constraints as they appear in the input PCF file. If the constraint is met, the report simply states the number of items scored by TRACE, reports no timing errors detected, and issues a brief report line, indicating important information (for example, the maximum delay for the particular constraint). If the constraint is not met, it gives the number of items scored by TRACE, the number of errors encountered, and a detailed breakdown of the error.

For errors in which the path delays are broken down into individual net and component delays, the report lists each physical resource and the logical resource from which the physical resource was generated.

As in the other three types of reports, descriptive material appears at the top. A timing summary always appears at the end of the reports.

The following sample error report (error.twr) represents the output generated with this TRACE command:

```
trce –e 3 ramb16_s1.ncd clkperiod.pcf –o error_report.twr
```

```
--------------------------------------------------------------------
Release 4
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

trce -e 3 ramb16_s1.ncd clkperiod.pcf -o error_report.twr

Design file:              ramb16_s1.ncd
Physical constraint file: clkperiod.pcf
Device,speed:             xc2v250,-5 (ADVANCED 1.84 2001-05-09)
Report level:             error report
--------------------------------------------------------------------



====================================================================
Timing constraint: TS01 = PERIOD TIMEGRP "clk" 10.333ns ;

 0 items analyzed, 0 timing errors detected.
--------------------------------------------------------------------

====================================================================
Timing constraint: OFFSET = IN 3.0 ns AFTER COMP "clk" TIMEGRP
"rams" ;

 18 items analyzed, 0 timing errors detected.
 Maximum allowable offset is   9.224ns.
--------------------------------------------------------------------

====================================================================
Timing constraint: TS02 = MAXDELAY FROM TIMEGRP "rams" TO TIMEGRP
"pads" 8.0 nS  ;

 1 item analyzed, 1 timing error detected.
 Maximum delay is   8.587ns.
--------------------------------------------------------------------
Slack:                  -0.587ns (requirement - data path)
  Source:               RAMB16.A
  Destination:          d0
  Requirement:          8.000ns
  Data Path Delay:      8.587ns (Levels of Logic = 2)
  Source Clock:         CLK rising at 0.000ns

  Data Path: RAMB16.A to d0
  Location      Delay type    Delay(ns)   Physical Resource
                                          Logical Resource(s)
  -------------------------------------   ------------------
  RAMB16.DOA0   Tbcko           3.006     RAMB16
                                          RAMB16.A
  IOB.O1        net           e 0.100     N$41
                (fanout=1)
  IOB.PAD       Tioop           5.481     d0
                                          I$22
                                          d0
  -------------------------------------   ------------------
  Total                         8.587ns (8.487ns logic, 0.100ns
                                          route)
                                        (98.8% logic, 1.2%
                                          route)

-----------------------------------------------------------------

1 constraint not met.

Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Setup/Hold to clock clk
---------------+------------+------------+
               |  Setup to  |  Hold to   |
Source Pad     | clk (edge) | clk (edge) |
---------------+------------+------------+
ad0            |   -0.013(R)|    0.325(R)|
ad1            |   -0.013(R)|    0.325(R)|
ad10           |   -0.013(R)|    0.325(R)|
ad11           |   -0.013(R)|    0.325(R)|
ad12           |   -0.013(R)|    0.325(R)|
ad13           |   -0.013(R)|    0.325(R)|
.
.
.
---------------+------------+------------+
Clock clk to Pad
---------------+------------+
               | clk (edge) |
Destination Pad|   to PAD   |
---------------+------------+
d0             |    9.563(R)|
---------------+------------+


Timing summary:
---------------

Timing errors: 1  Score: 587

Constraints cover 19 paths, 0 nets, and 21 connections (100.0%
coverage)

Design statistics:
   Maximum path delay from/to any node:   8.587ns
   Maximum input arrival time after clock:   9.224ns

Analysis completed Mon Jun 25 17:47:21 2001
--------------------------------------------------------------------
```

Verbose Report

The verbose report is similar to the error report, providing details on delays for all constrained paths and nets in the design. Entries are ordered by constraint and, within constraints, by slack, with a negative slack indicating an error condition. The maximum number of items listed for each constraint is set by the limit you enter on the command line.

Note The data sheet report and STAMP model display skew values on non-dedicated clock resources that do not display in the default period analysis of the normal verbose report. The data sheet report and STAMP model must include skew because skew affects the external timing model. To display skew values in the verbose report, use the –skew option.

The verbose report also contains a list of all time groups defined in the PCF file, and all of the members defined within each group.

The body of the verbose report enumerates each constraint as it appears in the input physical constraints file, the number of items scored by TRACE for that constraint, and the number of errors detected for the constraint. Each item is described, ordered by descending slack. A Report line for each item provides important information, such as the amount of delay on a net and by how much the constraint is met.

For path constraints, if there is an error, the report indicates the amount by which the constraint is exceeded. For errors in which the path delays are broken down into individual net and component delays, the report lists each physical resource and the logical resource from which the physical resource was generated.

If there are no errors, the report indicates that the constraint passed and by how much. Each logic and route delay is analyzed, totaled, and reported.

The following sample verbose report (verbose.twr) represents the output generated with this TRACE command:

trce –v 1 ramb16_s1.ncd clkperiod.pcf –o verbose_report.twr

```
------------------------------------------------------------------
Release 4
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

trce -v 1 ramb16_s1.ncd clkperiod.pcf -o verbose_report.twr

Design file:              ramb16_s1.ncd
Physical constraint file: clkperiod.pcf
Device,speed:             xc2v250,-5 (ADVANCED 1.84 2001-05-09)
Report level:               verbose report, limited to 1 item per
constraint
------------------------------------------------------------------


==================================================================
Timing constraint: TS01 = PERIOD TIMEGRP "clk" 10.333ns ;

 0 items analyzed, 0 timing errors detected.
------------------------------------------------------------------
```

```
================================================================================
Timing constraint: OFFSET = IN 3.0 ns AFTER COMP "clk" TIMEGRP
"rams" ;

 18 items analyzed, 0 timing errors detected.
 Maximum allowable offset is   9.224ns.
--------------------------------------------------------------------------------
Slack:                  6.224ns (requirement - (data path - clock
path - clock arrival))
  Source:               ssr
  Destination:          RAMB16.A
  Destination Clock:    CLK rising at 0.000ns
  Requirement:          7.333ns
  Data Path Delay:      2.085ns (Levels of Logic = 2)
  Clock Path Delay:     0.976ns (Levels of Logic = 2)

  Data Path: ssr to RAMB16.A
  Location      Delay type    Delay(ns)   Physical Resource
                                          Logical Resource(s)
  ------------------------------------    ------------------
  IOB.I         Tiopi         0.551       ssr
                                          ssr
                                          I$36
  RAM16.SSRA    net         e 0.100       N$9
                (fanout=1)
  RAM16.CLKA    Tbrck         1.434       RAMB16
                                          RAMB16.A
  ------------------------------------    ------------------
  Total                         2.085ns   (1.985ns logic, 0.100ns
                                          route)
                                          (95.2% logic, 4.8%
                                          route)


  Clock Path: clk to RAMB16.A
  Location      Delay type    Delay(ns)   Physical Resource
                                          Logical Resource(s)
  ------------------------------------    ------------------
  IOB.I         Tiopi         0.551       clk
                                          clk
                                          clk/new_buffer
  BUFGMUX.I0    net         e 0.100       clk/new_buffer
```

```
                (fanout=1)
  BUFGMUX.O     Tgi0o         0.225       I$9
                                          I$9
  RAM16.CLKA    net         e 0.100       CLK
                (fanout=1)

  -------------------------------------   -----------------
  Total                           0.976ns   (0.776ns logic, 0.200ns
                                          route)
                                          (79.5% logic, 20.5%
                                          route)

------------------------------------------------------------------

==================================================================
Timing constraint: TS02 = MAXDELAY FROM TIMEGRP "rams" TO TIMEGRP
"pads" 8.0 nS  ;

 1 item analyzed, 1 timing error detected.
 Maximum delay is   8.587ns.
------------------------------------------------------------------
Slack:                  -0.587ns (requirement - data path)
  Source:               RAMB16.A
  Destination:          d0
  Requirement:          8.000ns
  Data Path Delay:      8.587ns (Levels of Logic = 2)
  Source Clock:         CLK rising at 0.000ns

  Data Path: RAMB16.A to d0
   Location          Delay type       Delay(ns)  Physical Resource
                                             Logical Resource(s)
    -----------------------------------------------  -----------
    RAMB16.DOA0          Tbcko                 3.006   RAMB16
                                                       RAMB16.A
    IOB.O1               net (fanout=1)     e  0.100   N$41
    IOB.PAD              Tioop                 5.481   d0
                                                       I$22
                                                       d0
    -----------------------------------------------  -----------
    Total                                         8.587ns (8.487ns
logic, 0.100ns route)
                                          (98.8% logic, 1.2% route)
```

```
-------------------------------------------------------------------

1 constraint not met.

Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Setup/Hold to clock clk
---------------+------------+------------+
               |  Setup to  |  Hold to   |
Source Pad     | clk (edge) | clk (edge) |
---------------+------------+------------+
ad0            |   -0.013(R)|    0.325(R)|
ad1            |   -0.013(R)|    0.325(R)|
ad10           |   -0.013(R)|    0.325(R)|
ad11           |   -0.013(R)|    0.325(R)|
.
.
.
---------------+------------+------------+

Clock clk to Pad
---------------+------------+
               | clk (edge) |
Destination Pad|   to PAD   |
---------------+------------+
d0             |    9.563(R)|
---------------+------------+


Timing summary:
---------------

Timing errors: 1  Score: 587

Constraints  cover  19  paths,  0  nets,  and  21  connections  (100.0%
coverage)

Design statistics:
   Maximum path delay from/to any node:   8.587ns
   Maximum input arrival time after clock:   9.224ns


Analysis completed Mon Jun 25 17:57:24 2001
-----------------------------------------------------------------
```

TSI Report

The TSI (TimeSpec Interaction) report describes interactions among timing constraints in your design. The report lists three categories of constraints: exclusive, duplicate, and extracted.

- **Exclusive** coverage occurs when a particular constraint is the *only* one that covers some paths from, to, or between the start and end points listed in the report.
- **Duplicate** coverage occurs when a constraint covers some paths from, to, or between the start and end points, but one or more other constraints cover the same paths as well.
- **Extracted** coverage occurs when some paths from, to, or between the start and end points that were supposed to be covered by a particular constraint are instead covered by a higher priority constraint.

The TSI report does not list the paths themselves; it lists the start points as sources and the end points as destinations. You might see sources listed without destinations. For example a source can be listed in the exclusive constraint category without any destination. This means paths from that source are covered in the exclusive constraints category. However, the destination is covered by another category, duplicate or extracted. Similarly, you might see destinations listed without sources. This means paths to that destination are covered by one category of constraints, but the source is covered by another. Also note that not every source has a corresponding destination. Paths do not always exist between start and end points. The report lists start and end points between which there *may* be paths.

Two design examples with sample TSI reports follow. Design Example 1 shows how the timing tools determine extracted coverage. Design Example 2 shows how the timing tools determine duplicate coverage.

Extracted Coverage TSI Report Example

This design example shows extracted coverage. The following figure contains two source registers, S1 and S2, two destination registers D1 and D2, and common intermediate logic.

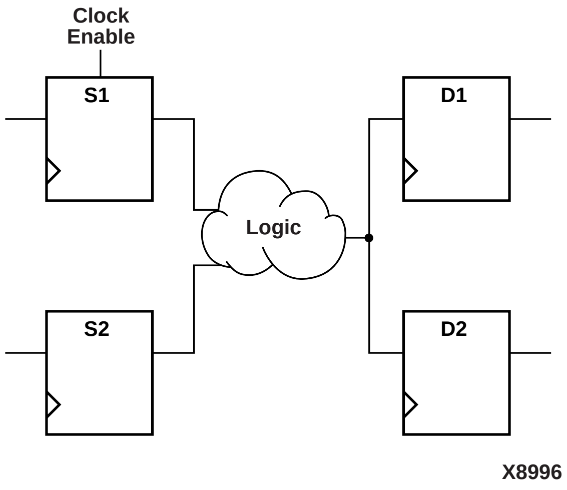


Figure 11-7 Extracted Coverage TSI Report Example

In this design, the registers S1, S2, D1 and D2 are all clocked by the same clock signal “CLOCK.” A combinatorial function in the logic is the result of the two signals driven by registers S1 and S2. The four circuit paths in this example are as follows:

A={S1->D1}

B={S1->D2}

C={S2->D1}

D={S2->D2}

The following excerpt from the PCF file shows the related design constraints. A period constraint is defined for the clock signal, and a maximum delay constraint for the paths originating from the source register S1. (S1 has a clock enable signal.)

```
net “CLOCK” period=10

from BEL “S1” maxdelay=15
```

The period constraint defines the paths A, B, C and D in the design example. The maximum delay constraint defines the paths A and B. Since the maximum delay constraint takes priority over the period constraint, the paths A and B are covered by the maximum delay constraint only; they are *extracted* from period constraint coverage. The paths C and D *are* covered by the period constraint.

The following TSI report for this design represents the output generated with this TRACE command:

trce –tsi example1.tsi tsinew.ncd example1.pcf –o example1.twr

```
------------------------------------------------------------------
Xilinx TRACE
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Design file:              tsi.ncd
Physical constraint file: example1.pcf
Report level:             timespec interaction report
------------------------------------------------------------------

==================================================================
Timing constraint: from BEL "S1" maxdelay = 15
==================================================================

Paths from, to, or between the following sources and destinations
are covered exclusively by this timing constraint:
     Sources:
        S1
     Destinations:

The following constraints duplicately cover some or all paths from,
to, or between the sources and destinations listed below each
constraint:

  Constraint: net "CLOCK" period = 10
     Sources:
     Destinations:
        D1  D2

The following constraints extracted some or all paths from, to, or
between the sources and destinations listed below each constraint:
  None


==================================================================
Timing constraint: net "CLOCK" period = 10
==================================================================

Paths from, to, or between the following sources and destinations
are covered exclusively by this timing constraint:
     Sources:
        S2
     Destinations:

The following constraints duplicately cover some or all paths from,
to, or between the sources and destinations listed below each
constraint:

  Constraint: from BEL "S1" maxdelay = 15
     Sources:
     Destinations:
        D1  D2

The following constraints extracted some or all paths from, to, or
between the sources and destinations listed below each constraint:

  Constraint: from BEL "S1" maxdelay = 15
     Sources:
        S1
     Destinations:

==================================================================


---------------------------------------------------
Pathtracing Controls: 12 entries (Default Settings)
---------------------------------------------------

Standard Name: reg_sr_q       State: Disabled
        Trio   Trlat  Trri   Trpo

Standard Name: reg_sr_clk     State: Disabled
        Trck  Tckr
```

```
Standard Name: lat_d_q       State: Disabled
        Tpli        Tplic     Tpmli       Tpmlic      Tpdli
        Tpdlic      Tpsli     Tpslic      Tpdsli      Tpdslic
        Tito        Tihto     Tsumc+Tito  Tsumc+Tihto Topc+Tito
        Topc+Tihto  Tasc+Tito Tasc+Tihto  Tinc+Tito   Tinc+Tihto
        Tdto        Tdtot     Thh0to      Thh1to      Thh2to
        Tcto        Twto      Twtot       Twtos       Twtots
        Twtods

.
.
.
.

---------------------------
Active Pathtracing Controls
---------------------------
Disabled Signals (global):
   None
Disabled Pins (global):
   None
Disabled Delays (global):
        Trio        Trlat       Trri        Trpo
        Trck        Tckr        Tpli        Tplic
        Tpmli       Tpmlic      Tpdli       Tpdlic
        Tpsli       Tpslic      Tpdsli      Tpdslic
        Tito        Tihto       Tsumc+Tito  Tsumc+Tihto
        Topc+Tito   Topc+Tihto  Tasc+Tito   Tasc+Tihto
.
.
.

Constraint Disables:
   None

-------------------------------------
Active Component Pathtracing Controls
-------------------------------------

Timespec interaction analysis completed Tue Jul 10 14:27:49 2001
------------------------------------------------------------------
```

Duplicate Coverage TSI Report Example

This example shows duplicate coverage. It uses a different PCF file with the same design as the previous example.

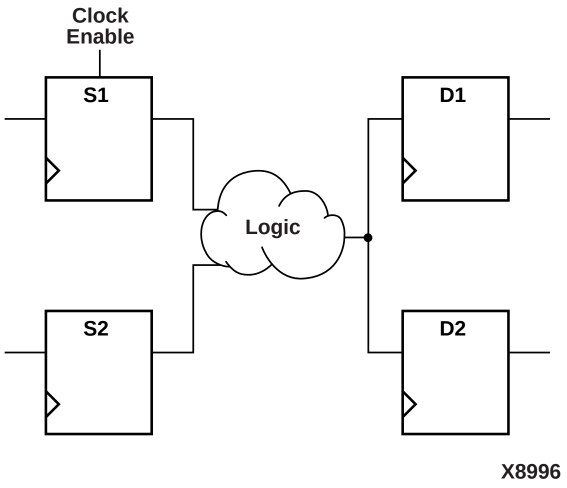


Figure 11-8 Duplicate Coverage TSI Report Example

To summarize the design, the registers S1, S2, D1 and D2 are all clocked by the same clock signal “CLOCK.” A combinatorial function in the logic is the result of the two signals driven by registers S1 and S2. The four circuit paths in the example are as follows:

A={S1->D1}

B={S1->D2}

C={S2->D1}

D={S2->D2}

In this example, the PCF file defines a maxdelay constraint for the paths from the source register S1 to the destination register D1.

```
net “CLOCK” period=10
from BEL “S1” to BEL “D1” maxdelay=15
```

The period constraint defines the paths A, B, C and D. The maxdelay constraint defines the set of paths A. The maxdelay constraint takes priority over the period constraint, but the timing tools are not path based and therefore cannot give precedence to the maxdelay constraint.

Note If precedence were given to the maxdelay constraint, the source S1 and destination D1 would have to be removed from coverage by the period constraint. This would also entail removing the paths defined by B and C. Since this is not the purpose of the constraint, the timing tools do not extract the paths affected by the period constraint.

In this situation, path A has duplicate coverage by both the period and the maxdelay constraints. The paths B, C, and D are covered by the period constraint alone.

Duplicate coverage may indicate that you need to refine your constraints. The following information may help you adjust your constraints:

- If the Clock Enable is used only on the S1 register, why doesn't the maxdelay constraint apply to all paths from S1 (as opposed to paths to D1)?
- If the S1 to D1 path is truly an exception to the period constraint, why isn't the S2 to D1 path also an exception?
- Define a clock period constraint using a timegrp (S1, D1). Include only S1, D1 and a slow period in the timegrp period constraint.
- Define another timegrp (S2, D2) period constraint listing only S2, D2 and a fast period.
- Define a constraint with any cross-group timing requirement.

The following TSI report for this design represents the output generated with this TRACE command:

```
trce –tsi example2.tsi tsinew.ncd example2.pcf –o example2.twr
```

```
------------------------------------------------------------------

Xilinx TRACE
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Design file:              tsi.ncd
Physical constraint file: example2.pcf
Report level:             timespec interaction report
------------------------------------------------------------------
==================================================================
Timing constraint: from BEL "S1" to BEL "D1" maxdelay = 15
==================================================================

Paths from, to, or between the following sources and destinations
are covered exclusively by this timing constraint:

  None

The following constraints duplicately cover some or all paths from,
to, or between the sources and destinations listed below each
constraint:

  Constraint: net "CLOCK" period = 10
     Sources:
        S1
     Destinations:
        D1

The following constraints extracted some or all paths from, to, or
between the sources and destinations listed below each constraint:

  None

==================================================================
Timing constraint: net "CLOCK" period = 10
==================================================================

Paths from, to, or between the following sources and destinations
are covered exclusively by this timing constraint:
     Sources:
        S2
     Destinations:
        D2

The following constraints duplicately cover some or all paths from,
to, or between the sources and destinations listed below each
constraint:

  Constraint: from BEL "S1" to BEL "D1" maxdelay = 15
     Sources:
        S1
     Destinations:
        D1

The following constraints extracted some or all paths from, to, or
between the sources and destinations listed below each constraint:

  None

==================================================================

---------------------------------------------------
Pathtracing Controls: 12 entries (Default Settings)
---------------------------------------------------

Standard Name: reg_sr_q       State: Disabled
        Trio   Trlat  Trri   Trpo

Standard Name: reg_sr_clk     State: Disabled
        Trck  Tckr

Standard Name: lat_d_q        State: Disabled
  Tpli          Tplic       Tpmli       Tpmlic        Tpdli
  Tpdlic        Tpsli       Tpslic      Tpdsli        Tpdslic
  Tito          Tihto       Tsumc+Tito  Tsumc+Tihto   Topc+Tito
  Topc+Tihto    Tasc+Tito   Tasc+Tihto  Tinc+Tito     Tinc+Tihto
.
.
.
---------------------------
Active Pathtracing Controls
---------------------------
Disabled Signals (global):
   None
```

```
Disabled Pins (global):
   None
Disabled Delays (global):
        Trio          Trlat         Trri          Trpo
        Trck          Tckr          Tpli          Tplic
        Tpmli         Tpmlic        Tpdli         Tpdlic
.
.
.
Constraint Disables:
   None

-------------------------------------
Active Component Pathtracing Controls
-------------------------------------

Timespec interaction analysis completed Tue Jul 10 14:22:56 2001
------------------------------------------------------------------
```

Halting TRACE

To halt TRACE, enter Ctrl C (on a workstation) or Ctrl-BREAK (on a PC). On a workstation, make sure that when you enter Ctrl C, the active window is the window from which you invoked TRACE. The program prompts you to confirm the interrupt. Some files may be left when TRACE is halted (for example, a TRACE report file or a physical constraints file), but these files may be discarded because they represent an incomplete operation.

Chapter 12

Speedprint

Speedprint is compatible with the following families.

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA

This chapter contains the following sections.

Speedprint Overview

The Speedprint program lists block delays for a device's speed grade. This program supplements data sheets, but does not replace them.

Note For additional information on block delays, see the *The Programmable Logic Data Book* or the data sheets at the Xilinx Web site.

The following figure shows the Speedprint design flow:

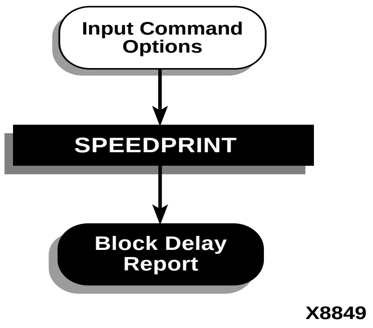


Figure 12-1 Speedprint

Speedprint Syntax

Use the following syntax to run speedprint:

speedprint [*options*] *device_name*

options can be any number of the Speedprint options listed in the "Speedprint Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

Speedprint Options

This section describes the options to the Speedprint command.

–min (Display Minimum Speed Data)

The –min option displays minimum speed data for a device. This option overrides the –s option if both are used.

–s (Speed Grade)

–s [*speed_grade*]

The –s option with a speed_grade argument (for example, -4) displays data for the specified speed grade. If the –s option is omitted, delay data for the default, which is the fastest speed grade, is displayed.

–t (Specify Temperature)

–t *temperature*

The –t option specifies the operating die temperature in degrees Celsius. If this option is omitted, the worst-case temperature is used.

–v (Specify Voltage)

–v *voltage*

The –v option specifies the operating voltage of the device in volts. If this option is omitted, the worst-case voltage is used.

Speedprint Example Commands

The following table describes some example commands:

Command	Description
speedprint	Prints usage message
speedprint xc4044xl	Uses the default speed grade
speedprint –s -3 xc4044xl speedprint -s 3 xc4044xl	Both displays block delays for speed grade -3
speedprint –v 3.0 -t 40 xc4044xl	Uses default speed grade at 3.0 volts and 40 degrees C
speedprint -min xc4044xl	Displays data for the minimum speed grade

Speedprint Example Reports

Following is a portion of a speed grade report for a Virtex device. The following command generates the displayed report:

```
speedprint v100e
```

Note the new section at the end of the report. This section provides adjustments for the I/O values in the speed grade report according to the I/O standard you are using. These adjustments model the delay reporting in the data sheet. To use these numbers, add the adjustment for the standard you are using to the delays reported for the IOBs.

The default speed grade, temperature, and voltage settings are described at the beginning of the file.

```
Block delay report for a: xv100e
              Speed grade is: -8
Version id for speed file is: PRELIMINARY 1.60 2001-06-06 xilinx

This speed file supports voltage adjustments over the range of
1.700000 to 1.900000 volts.
Temperature adjustments are supported over the junction temperature
range of -40.000000 to 85.000000 degrees Celsius
Derated delay values for operating conditions within these limits
are available using this speed file.

This report prepared for default temperature and voltage.

Note - this report is intended to present the effect of different
speedgrades and voltage/temperature adjustments on block delays,
for specific situations use the timing analyzer report instead.

Delays are reported in picoseconds, where a range of delays is given
they represent the fastest and slowest paths reported under that
name.

When a block is placed in a site normally used for another type of
block, a IOB placed in a Clock IOB site for example, small variations
in delay may occur which are not included in this report

External Setup and Hold requirements for global clocks
Tphf        -400         Tphfd         0         Tpsf        1500
Tpsfd       1800


Delays for a BLOCKRAM
Tback        831         Tbcka         0         Tbckd          0
Tbcke      -1097         Tbcko      2453         Tbckr       -961
Tbckw       -866         Tbdck       831         Tbeck       1928
Tbpwh       1164         Tbpwl      1164         Tbrck       1792
Tbwck       1697         Tgsrq      7531         Trpw       10100


Delays for a IOB
Tch       1116      Tcl       1116      Tgsrq     7531
Tgts      4050      Tiockice    1       Tiockiq   330 -  337
Tiockisr  -482      Tiocko    -573      Tiockoce  1
```

```
Tiockon 2736 - 2743 Tiockosr  -525      Tiockp    2335 - 2342
Tiockt    -238      Tiocktce  -50       Tiocktsr  -481
Tioiceck    546     Tioickp   -985      Tioickpd    -2501
Tiooceck    546     Tioock    845       Tioolp      2872
Tioop       2473    Tiopi     744       Tiopick     1258
Tiopickd    2772    Tiopid    944       Tiopli      1385
.
.
.
.

I/O numbers in this report should be adjusted according to the I/O
standard being used. The adjustments are as follows:
                                            Input          Output
 Standard Name     Slew    Drive    Adjustment     Adjustment
 =============     ====    =====    ==========     ==========
 LVTTL              2      FAST              0          13001
 LVTTL              4      FAST              0           5201
 LVTTL              6      FAST              0           3001
 LVTTL              8      FAST              0            902
 LVTTL              12     FAST              0              0
 LVTTL              16     FAST              0            -50
 LVTTL              24     FAST              0           -200
 LVTTL              2      SLOW              0          14601
 LVTTL              4      SLOW              0           7401
 LVTTL              6      SLOW              0           4701
 LVTTL              8      SLOW              0           2902
.
.
.
.
```

Chapter 13

BitGen

BitGen is compatible with the following families:

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA

This chapter contains the following sections:

- “BitGen Overview”
- “BitGen Syntax”
- “BitGen Input Files”
- “BitGen Output Files”
- “BitGen Options”

BitGen Overview

BitGen produces a bitstream for Xilinx device configuration. After the design has been completely routed, it is necessary to configure the device so that it can execute the desired function. This is done using files generated by BitGen, the Xilinx bitstream generation program. BitGen takes a fully routed NCD (Circuit Description) file as its input and produces a configuration bitstream—a binary file with a .bit extension.

The BIT file contains all of the configuration information from the NCD file defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. The binary data in the BIT file can then be downloaded into the FPGA’s memory cells or it can be used to create a PROM file (see the “PROMGen” chapter).

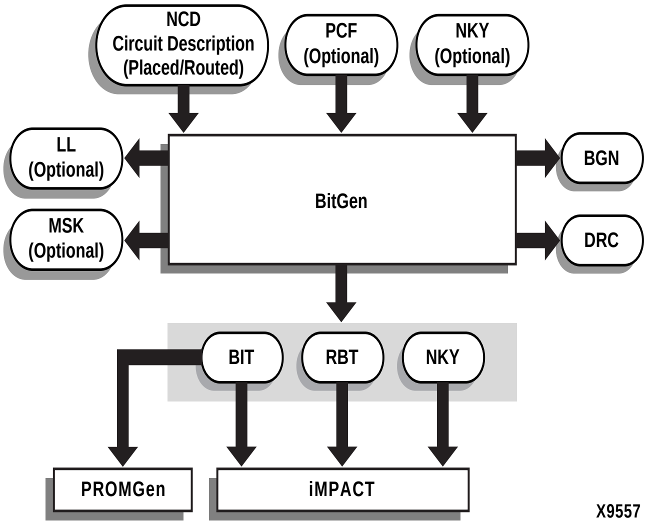


Figure 13-1 BitGen

BitGen Syntax

The following syntax creates a bitstream from your NCD file:

bitgen [*options*] *infile*[**.ncd**] [*outfile*] [*pcf_file*]

options is one or more of the options listed in the "BitGen Options" section.

infile is the name of the NCD design for which you want to create the bitstream. You may specify only one design file, and it must be the first file specified on the command line.

You do not have to use an extension. If you do not, .ncd is assumed. If you do use an extension, it must be .ncd.

outfile is the name of the output file. If you do not specify an output file name, BitGen creates a .bit file in your input file directory. If you specify any of the following options, the corresponding file is created in addition to the .bit file. If you do not specify an extension, BitGen appends the correct one for the specified option.

Option	Output File
-l	*outfile_name*.ll
-m	*outfile_name*.msk
-b	*outfile_name*.rbt

Pcf_file is the name of a physical constraints (PCF) file. BitGen uses this file to determine which nets in the design are critical for tiedown (see the "–t (Tie Unused Interconnect)" section). BitGen automatically reads the .pcf file by default. If the physical constraints file is the second file specified on the command line, it must have a .pcf extension. If it is the third file specified, the extension is optional; .pcf is assumed. If a .pcf file name is specified, it must exist, otherwise the input design name with a .pcf extension is read if that file exists.

A report file containing all of BitGen's output is automatically created under the same directory as the output file. The report file has the same root name as the output file and a .bgn extension.

BitGen Input Files

Input to BitGen consists of the following files:

- NCD file—a physical description of the design mapped, placed and routed in the target device. The NCD file must be fully routed.
- PCF—an optional user-modifiable ASCII Physical Constraints File. If you specify a PCF file on the BitGen command line, BitGen uses this file to determine which nets in the design are critical for tiedown (see the "–t (Tie Unused Interconnect)" section).
- NKY—an optional encryption key file.

Note For more information on encryption, see the following web site:

http://www.xilinx.com/products/virtex/handbook/

BitGen Output Files

Output from BitGen consists of the following files:

- BIT file—a binary file with a .bit extension. The BIT file contains all of the configuration information from the NCD file defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. The binary data in the BIT file can then be downloaded into the FPGA's memory cells or it can be used to create a PROM file (see the "PROMGen" chapter).
- RBT file—an optional "rawbits" file with an .rbt extension. The rawbits file consists of ASCII ones and zeros representing the data in the bitstream file. If you enter a –b option on the BitGen command line, an RBT file is produced in addition to the binary BIT file (see the "–b (Create Rawbits File)" section).
- LL file—an optional ASCII logic allocation file with an .ll extension. The logic allocation file indicates the bitstream position of latches, flip-flops, and IOB inputs and outputs. An LL file is produced if you enter a –l option on the BitGen command line (see the "–l (Create a Logic Allocation File)" section).
- MSK file—an optional mask file with an .msk extension. This file is used to compare relevant bit locations for executing a readback of configuration data contained in an operating FPGA. A MSK file is produced if you enter a –m option on the BitGen command line (see the "–m (Generate a Mask File)" section).
- BGN file—a report file containing information about the BitGen run.
- DRC file—a Design Rule Check (DRC) file for the design. A DRC runs and the DRC file is produced unless you enter a –d option on the BitGen command line (see the "–d (Do Not Run DRC)" section).
- NKY file—an encryption key file.

Note For more information on encryption, see the following web site:

http://www.xilinx.com/products/virtex/handbook/

BitGen Options

Following is a description of the command line options and how they affect the behavior of BitGen.

Note For a complete description of the Xilinx Development System command line syntax, see the "Command Line Syntax" section of the "Introduction" chapter.

–a (Tie All Interconnect)

Note This option supports Spartan, SpartanXL, and XC4000E / L / EX / XL / XLA.

Used with the –t option to force tiedown to fail if all nodes are not tied. This option also allows tiedown to implement user signals.

–b (Create Rawbits File)

Create a *rawbits* (*file_name*.rbt) file. The rawbits file consists of ASCII ones and zeros representing the data in the bitstream file.

If you are using a microprocessor to configure a single FPGA, you can include the rawbits file in the source code as a text file to represent the configuration data. The sequence of characters in the rawbits file is the same as the sequence of bits written into the FPGA.

–d (Do Not Run DRC)

Do not run DRC (Design Rule Check). Without the –d option, BitGen runs a DRC and saves the DRC results in two output files: the BitGen report file. (*file_name*.bgn) and the DRC file (*file_name*.drc). If you enter the –d option, no DRC information appears in the report file and no DRC file is produced.

Running DRC before a bitstream is produced detects any errors that could cause the FPGA to malfunction. If DRC does not detect any errors, BitGen produces a bitstream file (unless you use the –j option described in the "–j (No BIT File)" section).

You cannot disable the DRC with the –d option if you have specified a –t (Tie Unused Interconnect) option. The DRC always runs if you specify –t.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–g (Set Configuration)

The –g option specifies the startup timing and other bitstream options for Xilinx FPGAs. The settings for the –g option depend on the design's architecture. These settings are described in the following sections:

- "–g (Set Configuration—XC4000 and Spartan / XL)"
- "–g (Set Configuration—Virtex / -E / -II / -II PRO and Spartan-II / -IIE Devices)"

–g (Set Configuration—XC4000 and Spartan/XL)

Note For Virtex and Spartan-II / -IIE –g options, see the "–g (Set Configuration—Virtex / -E / -II / -II PRO and Spartan-II / -IIE Devices)" section.

This option specifies the startup timing and other bitstream options for the XC4000E / L, XC4000EX / XL / XV, and Spartan / XL devices. Timing sequences are predefined startup defaults that use the following syntax:

```
bitgen -g timing_sequence
```

There are four valid startup sequences: Cclk_Nosync, Cclk_Sync, Uclk_Nosync, and Uclk_Sync. These startup sequences are described in the next section. For more information about startup timing, refer to *The Programmable Logic Data Book.*

The default startup sequence for the –g option is Cclk_Nosync. This startup sequence makes an XC4000 or Spartan / XL device compatible with an XC3X00 device that is set for early Done and late Reset. The default command is as follows:

```
bitgen -g cclk_nosync
```

The –g option has sub-options that represent settings you use to set the configuration for an XC4000 or Spartan / XL design. These options use the following syntax:

```
bitgen –g option:setting
```

For example, to enable Cyclic Redundancy Checking (CRC), use the following syntax:

```
bitgen –g crc:enable
```

Note When mixing devices, the one with the latest “finished point” should be the master. The master stops clocking when it reaches the finished point. See *The Programmable Logic Data Book* for more information.

Cclk_Nosync

This is the default startup sequence for the –g option. Selecting this sequence causes the following defaults to take effect:

StartupClk:	Cclk
SyncToDone:	No
DoneActive:	C1
OutputsActive:	C2
GSRInactive:	C3

This startup sequence makes an XC4000 or Spartan / XL device consistent with an XC3X00 device set for early Done and late Reset.

Cclk_Sync

Selecting this sequence causes the following defaults to take effect:

StartupClk:	Cclk
SyncToDone:	Yes
DoneActive:	C1
OutputsActive:	DI_PLUS_1
GSRInactive:	DI_PLUS_1

This startup sequence is the most consistent with the XC3X00 devices, since it synchronizes the release of GSR and I / Os to the external DoneIn signal. This startup sequence makes an XC4000 or Spartan /

XL device consistent with an XC3X00 device set for early Done and late Reset.

Uclk_Nosync

Selecting this sequence causes the following defaults to take effect:

StartupClk:	Useclk
SyncToDone:	No
DoneActive:	U2
OutputsActive:	U3
GSRInactive:	U4

This startup sequence makes XC4000 or Spartan/XL devices inconsistent with XC3X00 devices if they are in the same daisy chain, since the release of Done is synchronized to an external User Clock. There is no synchronization of I/Os or GSR to DoneIn.

Uclk_Sync

Selecting this sequence causes the following defaults to take effect:

StartupClk:	Userclk
SyncToDone:	Yes
DoneActive:	U2
OutputsActive:	D1_PLUS_1
GSRInactive:	D1_PLUS_2

This startup sequence makes XC4000 or Spartan/XL devices inconsistent with XC3X00 devices if they are in the same daisy chain, since the release of Done is synchronized to an external User Clock. I/Os and GSR are synchronous to the clocks following DoneIn.

When using Uclk_Sync or Uclk_Nosync, you must provide a user clock to finish the configuration sequence. Without a user clock the FPGA will not configure.

AddressLines

Determines the number of address lines (18 or 22) used for device configuration. The *22* setting activates four extra device pins as configuration address lines.

Architectures:	XC4000EX only (XC4000XL, XC4000XLA, and XC4000XV always have 22 active address lines)
Settings:	18, 22
Default	18

Binary

Creates a binary file with programming data only. Use this option to extract and view programming data.

Architectures:	XC4000E, XC4000EX, XC4000L, XC4000XL, XC4000XLA, XC4000XV, Spartan, SpartanXL
Settings:	No, Yes
Default:	No

BSCAN_Config

When disabled, BSCAN_Config inhibits the BSCAN-based configuration after the device is successfully configured. This feature allows board testing without the risk of reconfiguring XLA devices by toggling the TCK / TMS / TDI / TDO lines.

Architectures:	XC4000XLA, XC4000XV, SpartanXL
Settings:	Disable, Enable
Default:	Disable

BSCAN_Status

When enabled, BSCAN_Status allows direct sensing of the DONE configuration state after performing a BSCAN-based configuration. Previously, there was no direct method for determining if a BSCAN-based configuration was successful.

Architectures:	XC4000XLA, XC4000XV, SpartanXL
Settings:	Disable, Enable
Default:	Disable

5V_Tolerant_IO

If set to On, this option allows a 3.3V device circuitry to tolerate 5V operation. For any device that operates on a mixed circuit environment with 3.3V and 5V, ensure that On is set. For any circuitry that operates exclusively on 3.3V, such as in a laptop computer, set the option to Off. The Off option reduces power consumption.

Architectures:	XC4000XLA, XC4000XV, SpartanXL
Settings:	On, Off
Default:	On

ConfigRate

Selects the configuration clock rate. There are two choices: slow or fast. Slow is equivalent to 1 MHz, and fast is equivalent to 8 MHz (nominal).

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, and SpartanXL
Settings:	Slow, Fast
Default:	Slow

CRC

Enables or disables Cyclic Redundancy Checking (CRC) on a chip-by-chip basis during configuration.

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV, Spartan, and SpartanXL
Settings:	Enable, Disable
Default:	Enable

DoneActive

Selects the event that activates the FPGA Done signal. There are a maximum of four events that you can select from at one time. These events are Cclk edges or external (user) clock edges.

The actual options available at any time depend on the selections made for StartupClk and SyncToDone.

Architectures:	XC4000E/L, XC4000EX/XL/XV/XLA, Spartan, and SpartanXL
Settings:	C1 — first-Cclk rising edge after the length count is met. C2 — second-Cclk rising edge after the length count is met. C3 — third-Cclk rising edge after the length count is met. C4 — fourth-Cclk rising edge after the length count is met. U2 — second-valid-user-clock rising edge after C1. U3 — third-valid-user-clock rising edge after C1. U4 — fourth-valid-user-clock rising edge after C1.
Default:	C1

Valid settings for DoneActive are as follows:

StartupClk	SyncToDone	DoneActive
Cclk	Yes	C1, C2 or C3
Cclk	No	C1, C2, C3, or C4
UserClk	Yes	C1 or U2
UserClk	No	C1, U2, U3, or U4

DonePin

Enables or disables internal pull-up on the DONE pin. The Pullnone setting indicates there is no connection to the pull-up.

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, and SpartanXL
Settings:	Pullup, Pullnone
Default:	Pullup

ExpressMode

When enabled, ExpressMode creates a unique type of bitstream for configuration.

Architectures:	XC4000XLA, XC4000XV, SpartanXL
Settings:	Disable, Enable
Default:	Disable

GSRInactive

Selects the event that releases the internal set-reset to the latches and flip-flops. You can select one of nine events: a Cclk edge, an external (user) clock edge, or the external signal DoneIn. Only some of these events become options at one time depending on the combination of StartupClk and SyncToDone selected.

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, SpartanXL
Settings:	C2 — second-Cclk rising edge after the length count is met. C3 — third-Cclk rising edge after the length count is met. C4 — fourth-Cclk rising edge after the length count is met. U2 — second-valid-user-clock rising edge after C1 (first-Cclk rising edge after length count is met). U3 — third-valid-user-clock rising edge after C1 (first-Cclk rising edge after length count is met). U4 — fourth-valid-user-clock rising edge after C1 (first-Cclk rising edge after length count is met). DI — when the DoneIn signal goes High. DI_PLUS_1 — first Cclk or valid user clock rising edge (depending on selection of StartupClk) after DoneIn goes High. DI_PLUS_2 — second Cclk or valid user clock rising edge (depending on selection of StartupClk) after DoneIn goes High.
Default:	C3

Valid settings for GSRInactive are as follows:

StartupClk	SyncToDone	GSRInactive
Cclk	Yes	C2, C3, DI, or DI_PLUS_1
Cclk	No	C2, C3, or C4
UserClk	Yes	U2, DI, DI_PLUS_1, or DI_PLUS_2
UserClk	No	U2, U3, or U4

Input

Sets the input threshold level for IOBs.

Architectures:	XC4000E/L, XC4000EX, Spartan
Settings:	TTL, CMOS
Default:	TTL

LC_Alignment

The LC_Alignment option determines how length count is calculated to control when the device changes from configuration to user operation. The two methods of calculating length count, DONE Alignment and Length Count Alignment, are discussed in the Configuration section of the *The Programmable Logic Data Book*. The FPGA Configuration Guidelines Application Note also contains length count information.

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV, Spartan, SpartanXL
Settings:	Length, DONE
Default	Length

M0Pin

Adds a pull-up or a pull-down to the M0 (Mode 0) pin. Selecting one option enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV, Spar-tanXL
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullnone

M1 Pin

Adds a pull-up or a pull-down to the M1 (Mode 1) pin. Selecting one option enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullnone

M2Pin

Adds a pull-up or a pull-down to the M2 (Mode 2) pin. Selecting one option enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullnone

Output

Sets the output level for IOBs,

Architectures:	XC4000E/L, XC4000EX, Spartan
Settings:	TTL, CMOS
Default:	TTL

OutputsActive

Selects the event that releases the I/O from 3-state condition and turns the configuration related pins operational. There are a maximum of four events that you can select from at one time. These events are selected from a group of Cclk edges, a group of external (user) clock edges, and the external signal DoneIn. The actual options available at any time depend on the selections made for StartupClk and SyncToDone.

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, SpartanXL
Settings:	C2 — second-Cclk rising edge after the length count is met. C3 — third-Cclk rising edge after the length count is met. C4 — fourth-Cclk rising edge after the length count is met. U2 — second-valid-user-clock rising edge after C1 (first-Cclk rising edge after length count is met) U3 — third-valid-user-clock rising edge after C1 (first-Cclk rising edge after length count is met) U4 — fourth-valid-user-clock rising edge after C1 (first-Cclk rising edge after length count is met) DI — when the DoneIn signal goes High DI_PLUS_1 — first Cclk or valid user clock rising edge (depending on selection of StartupClk) after DoneIn goes High DI_PLUS_2 — second Cclk or valid user clock rising edge (depending on selection of StartupClk) after DoneIn goes High
Default:	C2

Valid settings for OutputsActive are as follows.

StartupClk	SyncToDone	OutputsActive
Cclk	Yes	C2, C3, DI, or DI_PLUS_1
Cclk	No	C2, C3, or C4
UserClk	Yes	U2, DI, DI_PLUS_1, or DI_PLUS_2
UserClk	No	U2, U3, or U4

PowerDown

Enables or disables internal pull-up on the PowerDown pin. The Pullnone setting indicates there is no connection to the pull-up.

Architectures:	SpartanXL
Settings:	Pullup, Pullnone
Default:	Pullup

ReadAbort

Enables or disables aborting the readback sequence during the readback sequence.

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, SpartanXL
Settings:	Enable, Disable
Default:	Disable

ReadCapture

Enables or disables readback of configuration bitstream.

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, SpartanXL
Settings:	Enable, Disable
Default:	Disable

ReadClk

Sets the readback clock to be CClk or to a user-supplied clock (from a net inside the FPGA that is connected to the ‘i’ pin of the RDCLK schematic block).

Architectures:	XC4000E / L, XC4000EX / XL / XLA / XV, Spartan, SpartanXL
Settings:	Cclk (pin—see Note), Rdbk (user-supplied)
Default:	Cclk

Note In modes where CClk is an output, the pin is driven by the internal oscillator.

StartupClk

Selects a user-supplied clock or the internal Cclk for controlling the post-configuration startup phase of the FPGA initialization

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV, Spartan, SpartanXL
Settings:	Cclk (pin—see Note), UserClk (user-supplied)
Default:	Cclk

Note In modes where Cclk is an output, the pin is driven by the internal oscillator.

SyncToDone

Synchronizes the I/O startup sequence to the external DoneIn signal.

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV, Spartan, SpartanXL
Settings:	Yes, No
Default:	No

TdoPin

Adds a pull-up, a pull-down, or neither to the TDO pin (Test Data Out for Boundary Scan). Selecting one option enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

Architectures:	XC4000E/L, XC4000EX/XL/XLA/XV, Spartan, SpartanXL
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullnone

–g (Set Configuration—Virtex/-E/-II/-II PRO and Spartan-II/-IIE Devices)

The –g option has sub-options that represent settings you use to set the configuration for a Virtex/-E/-II/-II PRO or Spartan-II/-IIE design. These options have the following syntax:

```
bitgen –g option:setting
```

For example, to enable Readback, use the following syntax:

```
bitgen –g Readback
```

The following sections describe the startup sequences for the –g option.

Binary

Creates a binary file with programming data only. Use this option to extract and view programming data. Any changes to the header will not affect the extraction process.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	No, Yes
Default:	No

CclkPin

Adds an internal pull-up to the Cclk pin. The Pullnone setting disables the pullup.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullnone, Pullup
Default:	Pullup

Compress

This option uses the multiple frame write feature in the bitstream to reduce the size of the bitstream, not just the .bit file. Using the Compress option does not guarantee that the size of the bitstream will shrink.

ConfigRate

Virtex/-E/-II and Spartan-II use an internal oscillator to generate the configuration clock, CCLK, when configuring in a master mode. Use the configuration rate option to select the rate for this clock.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	4, 5, 7, 8, 9, 10, 13, 15, 20, 26, 30, 34, 41, 45, 51, 55, 60
Default:	4

DCMShutdown

When DCMShutdown is enabled, the DCM (Digital Clock Manager) resets if the SHUTDOWN and AGHIGH commands are loaded into the configuration logic.

Architectures:	Virtex-II and Virtex-II PRO
Settings:	Disable, Enable
Default:	Disable

DebugBitstream

If the device does not configure correctly, you can debug the bitstream using the DebugBitstream option. A debug bitstream is significantly larger than a standard bitstream. The values allowed for the DebugBitstream option are No and Yes.

Note You should use this option only if your device is configured to use slave or master serial mode.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Values:	No, Yes

In addition to a standard bitstream, a debug bitstream offers the following features:

- Writes 32 0s to the LOUT register after the synchronization word
- Loads each frame individually

- Performs a cyclical redundancy check (CRC) after each frame
- Writes a column number to the LOUT register after each frame

DisableBandgap

Disables bandgap generator for DCMs to save power.

Architectures:	Virtex-II and Virtex-II PRO
Settings:	No, Yes
Default:	No

DONE_cycle

Selects the Startup phase that activates the FPGA Done signal. Done is delayed when DonePipe=Yes.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	1, 2, 3, 4, 5, 6
Default:	4

DonePin

Adds an internal pull-up to the DonePin pin. The Pullnone setting disables the pullup.

Use this option only if you are planning to connect an external pull-up resistor to this pin. The internal pull-up resistor is automatically connected if you do not use this option.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pullnone
Default:	Pullup

DonePipe

This option is intended for use with FPGAs being set up in a high-speed daisy chain configuration.When set to Yes, the FPGA waits on the CFG_DONE (DONE) pin to go High and then waits for the first clock edge before moving to the Done state.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	No, Yes
Default:	No

DriveDone

This option actively drives CFG_DONE (Done) high as opposed to using pullup.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	No, Yes
Default:	No

Encrypt

Encrypts the bitstream.

Architectures:	Virtex-II, Virtex-II PRO
Settings:	No, Yes
Default:	No

Note For more information on encryption, see the following web site:

http://www.xilinx.com/products/virtex/handbook/

Gclkdel0, Gclkdel1, Gclkdel2, Gclkdel3

Use these options to add delays to the global clocks. *You should not use this option unless instructed by Xilinx.*

Architectures:	Virtex/-E/-II, Spartan-II/-IIE
Settings:	11111, *binary string*
Default:	11111

GSR_cycle

Selects the Startup phase that releases the internal set-reset to the latches, flip-flops, and BRAM output latches. The Done setting releases GSR when the DoneIn signal is High. DoneIn is either the value of the Done pin or a delayed version if DonePipe=Yes

Architectures:	Virtex/-E, Spartan-II/-IIE
Settings:	Done, 1, 2, 3, 4, 5, 6, Keep
Default:	6

Keep should only be used when partial reconfiguration is going to be implemented. Keep prevents the configuration state machine from asserting control signals that could cause the loss of data.

GWE_cycle

Selects the Startup phase that asserts the internal write enable to flip-flops, LUT RAMs, and shift registers. It also enables the BRAMs. Before the Startup phase both BRAM writing and reading are disabled.The Done setting asserts GWE when the DoneIn signal is High. DoneIn is either the value of the Done pin or a delayed version if DonePipe=Yes. The Keep setting is used to keep the current value of the GWE signal

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Done, 1, 2, 3, 4, 5, 6, Keep
Default:	6

GTS_cycle

Selects the Startup phase that releases the internal 3-state control to the IO buffers. The Done setting releases GTS when the DoneIn signal is High. DoneIn is either the value of the Done pin or a delayed version if DonePipe=Yes

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Done, 1, 2, 3, 4, 5, 6, Keep
Default:	5

HswapenPin

Adds a pull-up, pull-down, or neither to the HSWAP_EN pin. The Pullnone option indicates there is no connection to either the pull-up or the pull-down.

Architectures:	Virtex-II
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

Key0, Key1, Key2, Key3, Key4, Key5

Sets Key*x* for bitstream encryption. The pick option causes BitGen to select a random number for the value.

Architectures:	Virtex-II, Virtex-II PRO
Settings:	Pick, *hex_string*
Default:	Pick

Note For more information on encryption, see the following web site:

http://www.xilinx.com/products/virtex/handbook/

KeyFile

Specifies the name of the input encryption file.

Architectures:	Virtex-II, Virtex-II PRO
Settings:	*string*

Keyseq0, Keyseq1, Keyseq2, Keyseq3, Keyseq4, Keyseq5

Sets the key sequence for key*x*. The settings are equal to the following:

- S=single
- F=first
- M=middle
- L=last

Architectures:	Virtex-II, Virtex-II PRO
Settings:	S, F, M, L
Default:	S

LCK_cycle

Selects the Startup phase to wait until DLLs lock. If NoWait is selected, the Startup sequence does not wait for DLLs.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	0,1, 2, 3, 4, 5, 6, NoWait
Default:	NoWait

M0Pin

The M0 pin is used to determine the configuration mode. Adds an internal pull-up, pull-down or neither to the M0 pin. The following settings are available. The default is PullUp. Select Pullnone to disable both the pull-up resistor and pull-down resistor on the M0 pin.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

M1Pin

The M1 pin is used to determine the configuration mode. Adds an internal pull-up, pull-down or neither to the M1 pin. The following settings are available. The default is PullUp.

Select Pullnone to disable both the pull-up resistor and pull-down resistor on the M1 pin.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

M2Pin

The M2 pin is used to determine the configuration mode. Adds an internal pull-up, pull-down or neither to the M2 pin. The default is PullUp. Select Pullnone to disable both the pull-up resistor and pull-down resistor on the M2 pin.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

Match_cycle

Specifies a stall in this Startup cycle until DCI (Digitally Controlled Impedance) match signals are asserted.

Architectures:	Virtex-II, Virtex-II PRO
Settings:	NoWait, 0, 1, 2, 3, 4, 5, 6
Default:	NoWait

Persist

This option is needed for Readback and Partial Reconfiguration using the SelectMAP configuration pins. If Persist is set to Yes, the pins used for SelectMAP mode are prohibited for use as user IO. Refer to the data sheet for a description of SelectMAP mode and the associated pins.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	No, Yes
Default:	No

ProgPin

Adds an internal pull-up to the ProgPin pin. The Pullnone setting -disables the pullup. The pull-up affects the pin after configuration.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pullnone
Default:	Pullnone

ReadBack

This option allows you to perform Readback by the creating the necessary bitstream.

Architectures:	Virtex/-E/-II/-II PRO, Spartan-II/-IIE

Security

Selecting Level1 disables Readback. Selecting Level2 disables Readback and Partial Reconfiguration.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	None, level1, Level2
Default:	None

StartCBC

Sets the starting CBC (Cipher Block Chaining) value. The pick option causes BitGen to select a random number for the value.

Architectures:	Virtex-II, Virtex-II PRO
Settings:	Pick, *hex_string*
Default:	Pick

StartKey

Sets the starting key number.

Architectures:	Virtex-II, Virtex-II PRO
Settings:	0, 3
Default:	0

StartupClk

The startup sequence following the configuration of a device can be synchronized to either Cclk, a User Clock, or the JTAG Clock. The default is Cclk.

- Cclk

  Enter Cclk to synchronize to an internal clock provided in the FPGA device.

- UserClk

  Enter UserClk to synchronize to a user-defined signal connected to the CLK pin of the STARTUP symbol.

- JTAG Clock

  Enter JtagClk to synchronize to the clock provided by JTAG. This clock sequences the TAP controller which provides the control logic for JTAG.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Cclk (pin—see Note), UserClk (user-supplied), JtagCLK
Default:	Cclk

Note In modes where Cclk is an output, the pin is driven by an internal oscillator.

TckPin

Adds a pull-up, a pull-down or neither to the TCK pin, the JTAG test clock. Selecting one setting enables it and disables the others. The

Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

TdiPin

Adds a pull-up, a pull-down, or neither to the TDI pin, the serial data input to all JTAG instructions and JTAG registers. Selecting one setting enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

TdoPin

Adds a pull-up, a pull-down, or neither to the TdoPin pin, the serial data output for all JTAG instruction and data registers. Selecting one setting enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

TmsPin

Adds a pull-up, pull-down, or neither to the TMS pin, the mode input signal to the TAP controller. The TAP controller provides the control logic for JTAG. Selecting one setting enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pullup

UnusedPin

Adds a pull-up, a pull-down, or neither to the UnusedPin, the serial data output for all JTAG instruction and data registers. Selecting one setting enables it and disables the others. The Pullnone setting indicates there is no connection to either the pull-up or the pull-down.

The following settings are available. The default is PullDown.

Architectures:	Virtex, Virtex-E, Virtex-II, Virtex-II PRO, Spartan-II, and Spartan-IIE
Settings:	Pullup, Pulldown, Pullnone
Default:	Pulldown

UserID

You can enter up to an 8-digit hexadecimal code in the User ID register. You can use the register to identify implementation revisions.

–j (No BIT File)

Do not create a bitstream file (.bit file). This option is generally used when you want to generate a report without producing a bitstream. For example, if you wanted to run DRC without producing a bitstream file, you would use the -j option.

Note The .msk or .rbt files may still be created.

–l (Create a Logic Allocation File)

This option creates an ASCII logic allocation file (*design*.ll) for the selected design. The logic allocation file indicates the bitstream position of latches, flip-flops, and IOB inputs and outputs.

In some applications, you may want to observe the contents of the FPGA internal registers at different times. The file created by the –l option helps you identify which bits in the current bitstream represent outputs of flip-flops and latches. Bits are referenced by frame and bit number within the frame.

The iMPACT tool uses the design.ll file to locate signal values inside a readback bitstream.

–m (Generate a Mask File)

Creates a mask file. This file is used to compare relevant bit locations for executing a readback of configuration data contained in an operating FPGA.

–n (Save a Tied design)

Note This option supports Spartan, SpartanXL, and XC4000E/L/EX/XL/XLA.

This command is used with the –t option (described below) to save the tied NCD file as *_file_name*.ncd (note the underscore in front of the file name). The tied design file is placed in the same directory as the output file. It has the same root name as the output file with an .ncd extension. If you do not specify an output file, the tied design file is placed in the input file's directory and is named *_file_name*.ncd, where *_file_name* is the root name of the input file. Use TRACE to run timing analysis on the tied design. You can also use the FPGA Editor to check the effects of the tiedown.

–t (Tie Unused Interconnect)

Note This option supports Spartan, SpartanXL, and XC4000E/L/EX/XL/XLA.

This option causes all unused interconnect to be tied to a logic low or to a known level, keeping internal noise and power consumption to a minimum. When you use the –t option, DRC runs first (before tiedown). BitGen terminates if any DRC error occurs. A DRC warning does not cause the bitstream generation program to abort, but it may cause tiedown to fail.

After DRC, the –t option performs the following functions:

- Ties all possible unused interconnect to tie sites or unused CLB outputs and configures those outputs with a logic low (F=0 or G=0)
- Attempts to tie any remaining interconnect to CLB outputs which have not been designated as critical
- Attempts to tie remaining interconnect to the global or to the auxiliary clock buffer outputs if unused (only in conjunction with the -a option)

The only condition under which tie will add interconnect to a "critical" net is if you use the –u option (allowing interconnect to be added to critical nets as a "last resort"). A "critical" net is one with a priority greater than 3.

The –t option does not add an XC4000 3-state buffer input (I) pin or 3-state (T) pin to a net.

When you add interconnect to used CLB or buffer outputs, delays may be added on any net to which the outputs are connected. To prevent the added delay, assign the net a priority greater than 3. You can do this through the physical constraints file or through the FPGA Editor. See the PRIORITIZE physical constraint in the *Constraints Guide*. Note that flagging too many nets as critical could cause the tiedown to fail. When an interconnect is tied to a user-defined net, you get a message giving the number of nodes added to the net. Delay characteristics for the net associated with that source may change. (Only in conjunction with the -a option)

When certain pins cannot be tied, you receive a warning message supplying information about the design's untied interconnect.

To remove the obstacles that have caused tiedown to fail, look carefully at nets close to an untied PIP. An input pin could have multiple input PIPs, and all of them could source the pin. If each of these PIPs is associated with a critical net, they are not used, and the input pin is left untied. To correct the problem, make one of the nets "non-critical." Do this by removing the PRIORITIZE constraint from the net in the PCF file or in the FPGA Editor. Then run TRACE (the timing analysis program) and evaluate any delay that might have been added to the net (only in conjunction with the -a option).

If you use the –n option, the tied design is saved in a file *_file_name*.ncd (note the underscore before the file name). You can load the file into the FPGA Editor and examine the results of tiedown. You can look at all of the original nets that have been affected by tiedown and the net delays before and after tiedown.

Like unused internal interconnect, unused external I/O pins on the chip must also have defined signal levels, that is, they must not be in a floating condition. In XC4000E/EX FPGAs, unused IOBs are automatically pulled HIGH with pull-up resistors.

Partial tiedown is the new default. Tiedown will print the number of untied nodes and then continue. See the -a option also. Partial tiedown *never* ties to user signals.

–u (Use Critical Nets Last)

Note This option supports Spartan, SpartanXL, and XC4000E/L/EX/XL/XLA.

Because of possible added delay, tiedown does not add interconnect to any net that has been assigned a priority greater than 3. This option allows interconnect to be added to critical nets as a “last resort.”

–w (Overwrite Existing Output File)

Enables you to overwrite an existing BIT, LL, MSK, or RBT output file.

Chapter 14

PROMGen

PROMGen is compatible with the following families:

- Virtex™ / -II / -II PRO / -E
- Spartan™ / -II / -IIE / XL
- XC4000™E / L / EX / XL / XLA

This chapter contains the following sections:

- "PROMGen Overview"
- "PROMGen Syntax"
- "PROMGen Input Files"
- "PROMGen Output Files"
- "PROMGen Options"
- "Bit Swapping in PROM Files"
- "PROMGen Examples"

PROMGen Overview

PROMGen formats a BitGen-generated configuration bitstream (BIT) file into a PROM format file. The PROM file contains configuration data for the FPGA device. PROMGen converts a BIT file into one of three PROM formats: MCS-86 (Intel), EXORMAX (Motorola), or TEKHEX (Tektronix). It can also generate a binary or hexadecimal file format.

The following figure shows the inputs and the possible outputs of the PROMGen program:

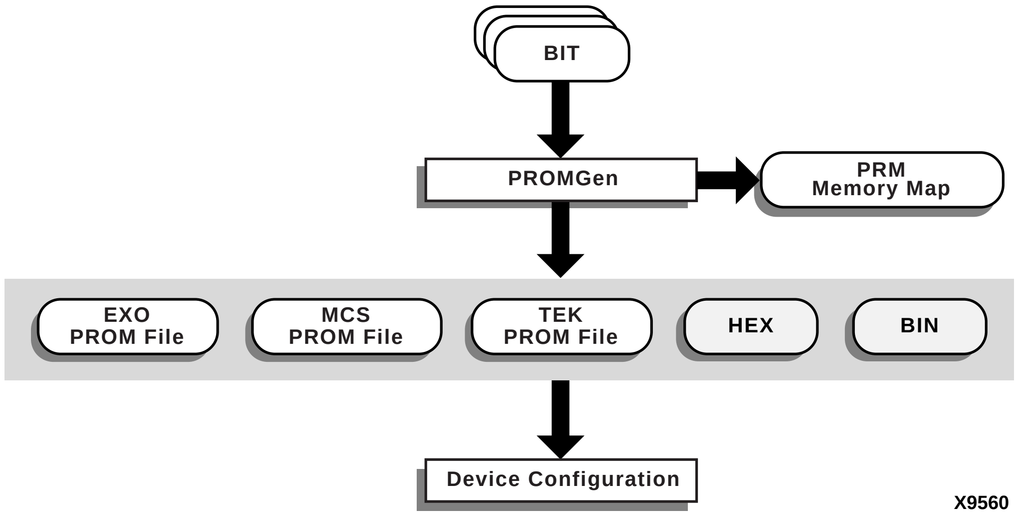


Figure 14-1 PROMGen

There are two functionally equivalent versions of PROMGen. There is a stand-alone version that you can access from an operating system prompt. There is also an interactive version, called the PROM File Formatter, that you can access from inside Project Navigator. This chapter first describes the stand-alone version; the interactive version is described in the *PROM File Formatter* online help.

You can also use PROMGen to concatenate bitstream files to daisy-chain FPGAs.

Note If the destination PROM is one of the Xilinx Serial PROMs, you are using a Xilinx PROM Programmer, and the FPGAs are not being daisy-chained, it is not necessary to make a PROM file.

PROMGen Syntax

To start PROMGen from the operating system prompt, use the following syntax:

```
promgen [options]
```

options can be any number of the options listed in the "PROMGen Options" section. Separate multiple options with spaces.

PROMGen Input Files

The input to PROMGEN consists of BIT files— one or more bitstream files. BIT files contain configuration data for an FPGA design.

PROMGen Output Files

Output from PROMGEN consists of the following files:

- PROM files—The file or files containing the PROM configuration information. Depending on the PROM file format your PROM programmer uses, you can output a TEK, MCS, BIN, or EXO file. If you are using a microprocessor to configure your devices, you can output a HEX file, which contains a hexadecimal representation of the bitstream.
- PRM file—The PRM file is a PROM image file. It contains a memory map of the output PROM file. The file has a .prm extension.

PROMGen Options

This section describes the options that are available for the PROMGen command.

–b (Disable Bit Swapping—HEX Format Only)

This option only applies if the –p option specifies a HEX file for the output of PROMGen. By default (no –b option), bits in the HEX file are swapped compared to bits in the input BIT files. If you enter a –b option, the bits are not swapped. Bit swapping is described in the "Bit Swapping in PROM Files" section.

–c (Checksum)

```
promgen –c
```

The –c option generates a checksum value appearing in the .prm file. This value should match the checksum in the prom programmer. Use this option to verify that correct data was programmed into the prom.

–d (Load Downward)

```
promgen -d hexaddress0 filename filename...
```

This option loads one or more BIT files from the starting address in a downward direction. Specifying several files after this option causes the files to be concatenated in a daisy chain. You can specify multiple –d options to load files at different addresses. You must specify this option immediately before the input bitstream file.

Here is the multiple file syntax.

```
promgen -d hexaddress0 filename filename...
```

Here is the multiple –d options syntax.

```
promgen -d hexaddress1 filename -d hexaddress2
filename...
```

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–l (Disable Length Count)

```
promgen –l
```

The –l option disables the length counter in the FPGA bitstream. It is valid only for SpartanXL, 4000EX, 4000XL, 4000XV, and 4000XLA devices. Use this option when chaining together bitstreams exceeding the 24 bit limit imposed by the length counter.

–n (Add BIT FIles)

```
–n file1[.bit] file2[.bit]...
```

This option loads one or more BIT files up or down from the next available address following the previous load. The first –n option *must* follow a –u or –d option because -n does not establish a direction. Files specified with this option are not daisy-chained to previous files. Files are loaded in the direction established by the nearest prior –u, –d, or –n option.

The following syntax shows how to specify multiple files. When you specify multiple files, PROMGen daisy-chains the files.

```
promgen –d hexaddress file0 –n file1 file2...
```

The syntax for using multiple –n options follows. Using this method prevents the files from being daisy-chained.

```
promgen –d hexaddress file0 –n file1 -n file2...
```

–o (Output File Name)

```
–o file1[.ext] file2[.ext]...
```

This option specifies the output file name of a PROM if it is different from the default. If you do not specify an output file name, the PROM file has the same name as the first BIT file loaded.

ext is the extension for the applicable PROM format.

Multiple file names may be specified to split the information into multiple files. If only one name is supplied for split PROM files (by you or by default), the output PROM files are named *file_#*.ext, where *file* is the base name, # is 0, 1, etc., and *ext* is the extension for the applicable PROM format.

```
promgen –d hexaddress file0 –o filename
```

–p (PROM Format)

```
–p {mcs | exo | tek | hex| bin}
```

This option sets the PROM format to MCS (Intel MCS86), EXO (Motorola EXORMAX), or TEK (Tektronix TEKHEX). The option may also produce a HEX file, which is a hexadecimal representation of the configuration bitstream used for microprocessor downloads. The default format is MCS.

The option may also produce a bin file, which is a binary representation of the configuration bitstream used for microprocessor downloads.

–r (Load PROM File)

```
–r promfile
```

This option reads an existing PROM file as input instead of a BIT file. All of the PROMGen output options may be used, so the –r option can be used for splitting an existing PROM file into multiple PROM files or for converting an existing PROM file to another format.

–s (PROM Size)

```
–s promsize1 promsize2...
```

This option sets the PROM size in kilobytes. The PROM size must be a power of 2. The default value is 64 kilobytes. The –s option must precede any –u, –d, or –n options.

Multiple *promsize* entries for the –s option indicates the PROM will be split into multiple PROM files.

Note PROMGen PROM sizes are specified in bytes. *The Programmable Logic Data Book* specifies PROM sizes in bits for Xilinx serial PROMs See the –x option for more information.

–u (Load Upward)

```
–u hexaddress0 filename1 filename2...
```

This option loads one or more BIT files from the starting address in an upward direction. When you specify several files after this option, PROMGen concatenates the files in a daisy chain. You can load files at different addresses by specifying multiple –u options.

This option must be specified immediately before the input bitstream file.

–w (Overwrite Existing Output File)

```
promgen –w
```

The –w option overwrites an existing output file, and *must* be used if an output file exists. If this option is not used, PROMGen issues an error.

–x (Specify Xilinx PROM)

–x *xilinx_prom1 xilinx_prom2...*

The –x option specifies one or more Xilinx serial PROMs for which the PROM files are targeted. Use this option instead of the –s option if you know the Xilinx PROMs to use.

Multiple *xilinx_prom* entries for the –x option indicates the PROM will be split into multiple PROM files.

Bit Swapping in PROM Files

PROMGen produces a PROM file in which the bits within a byte are swapped compared to the bits in the input BIT file. Bit swapping (also called “bit mirroring”) reverses the bits within each byte, as shown in the following figure:

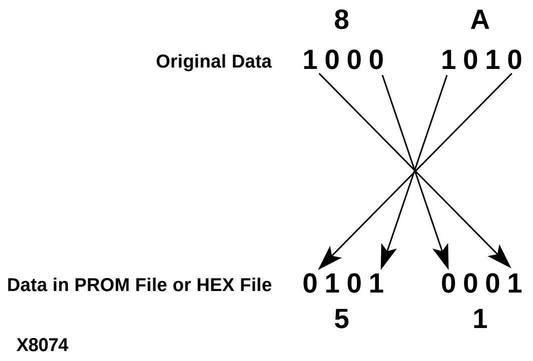


Figure 14-2 Bit Swapping

In a bitstream contained in a BIT file, the Least Significant Bit (LSB) is always on the left side of a byte. But when a PROM programmer or a microprocessor reads a data byte, it identifies the LSB on the right side of the byte. In order for the PROM programmer or microprocessor to read the bitstream correctly, the bits in each byte must first be swapped so they are read in the correct order.

In this release of the Xilinx Development System, the bits are swapped for all of the PROM formats: MCS, EXO, BIN, and TEK. For a HEX file output, bit swapping is on by default, but it can be turned off by entering a –b PROMGen option that is available only for HEX file format.

PROMGen Examples

To load the file test.bit up from address 0x0000 in MCS format, enter the following information at the command line:

```
promgen –u 0 test
```

To daisy-chain the files test1.bit and test2.bit up from address 0x0000 and the files test3.bit and test4.bit from address 0x4000 while using a 32K PROM and the Motorola EXORmax format, enter the following information at the command line:

```
promgen –s 32 –p exo –u 00 test1 test2 –u 4000 test3
test4
```

To load the file test.bit into the PROM programmer in a downward direction starting at address 0x400, using a Xilinx XC1718D PROM, enter the following information at the command line:

```
promgen –x xc1718d –d 0x400 test
```

To specify a PROM file name that is different from the default file name enter the following information at the command line:

```
promgen options filename –o newfilename
```

Chapter 15

IBISWriter

The IBISWriter program is compatible with the following families:

- Virtex / -II / -E
- Spartan / -II / XL
- XC4000E / XL / XLA
- CoolRunner XPLA3
- XC9500 / XL / XV

The chapter contains the following sections:

- "IBISWriter Overview"
- "IBISWriter Syntax"
- "IBISWriter Input Files"
- "IBISWriter Output Files"
- "IBISWriter Options"

IBISWriter Overview

The Input / Output Buffer Information Specification (IBIS) is a device modeling standard. IBIS allows for the development of behavioral models used to describe the signal behavior of device interconnects. These models preserve proprietary circuit information, unlike structural models such as those generated from SPICE (Simulation Program with Integrated Circuit Emphasis) simulations. IBIS buffer models are based on V / I curve data produced either by measurement or by circuit simulation.

IBIS models are constructed for each IOB standard, and an IBIS file is a collection of IBIS models for all I / O standards in the device. An IBIS file also contains a list of the used pins on a device that are bonded to IOBs configured to support a particular I/O standard (which associates the pins with a particular IBIS buffer model).

The IBIS standard specifies the format of the output information file, which contains a file header section and a component description section. The *Golden Parser* has been developed by the IBIS Open Forum Group (http://www.eigroup.org/ibis) to validate the resulting IBIS model file by verifying that the syntax conforms to the IBIS data format.

The IBISWriter tool requires a design source file as input. For FPGA designs, this is a physical description of the design in the form of an NCD (Native Circuit Description) file with a .ncd file extension. For CPLD designs, the input is produced by the CPLD fitter tools and has a .pnx file extension.

IBISWriter outputs a .ibs file. This file consists of a list of pins used by your design; the signals internal to the device that connect to those pins; and the IBIS buffer models for the IOBs connected to the pins.

Note To see an example of an IBIS file, refer to Virtex Tech Topic VTT004 at the following web location:

http://www.xilinx.com/products/virtex/techtopic/vtt004.pdf

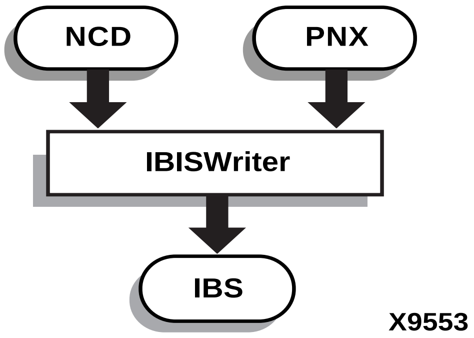


Figure 15-1 IBISWriter

IBISWriter Syntax

Use the following syntax to run IBISWriter from the command line:

```
ibiswriter [options] infile outfile[.ibs]
```

options is one or more of the options listed in the "IBISWriter Options" section.

infile is the design source file for the specified design. For FPGA designs, *infile* must have a .ncd extension. For CPLD designs, *infile* is produced by the CPLD fitter tools and must have a .pnx extension.

outfile[.ibs] is the destination for the design specific IBIS file. The .ibs extension is optional. The length of the IBIS file name, including the .ibs extension, cannot exceed 24 characters.

IBISWriter Input Files

IBISWriter requires a design source file as input.

- FPGA Designs

  Requires a physical description of the design in the form of an NCD (Native Circuit Description) file with a .ncd file extension

- CPLD Designs

  The input is produced by the CPLD fitter tools and has a .pnx file extension

IBISWriter Output Files

IBISWriter outputs a .ibs ASCII file. This file consists of a list of pins used by your design, the signals internal to the device that connect to those pins, and the IBIS buffer models for the IOBs connected to the pins. The format of the IBIS output file is determined by the IBIS standard. The output file must be able to be validated by the Golden Parser to ensure that the file format conforms to the specification.

IBISWriter Options

This section provides information on IBISWriter command line options.

–g (Set Reference Voltage)

The –g command line option varies by architecture as shown in the following table.

Use the following syntax when using the –g option:

```
ibiswriter –g option_value_pair infile outfile.ibs
```

The following is an example using the VCCIO:LVTTL option value pair.

```
ibiswriter –g VCCIO:LVTTL design.ncd design.ibs
```

Note The –g option supports only the architectures listed in the following table.

Table 15-1 –g Options

Architecture	Option	Value	Description
Virtex-E	OperatingConditions	Typical_Slow_Fast, Mixed	Use this option to set operating condition parameters. Typical_Slow_Fast refers to operating range defined by temperature, VCCIO, and manufacturing process ranges
XC4000XLA	5VTolerant	False, True	Use this option to indicate that an IOB has been configured to activate circuitry to allow the IOB (when used as an input) to go above the 3.3V Vcc rail
XC9500	VCCIO	LVTTL, TTL	Use this option to configure I/Os for 3.3V (LVTTL) or 5V (TTL) operation
XC9500XL	VCCIO	LVCMOS2, LVTTL	Use this option to configure outputs for 3.3V (LVTTL) or 2.5V (LVCMOS2) operation. Each user pin is compatible with 5V, 3.3V, and 2.5V inputs.

NGDAnno

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA

This chapter describes the NGDAnno program. The chapter contains the following sections:

NGDAnno Overview

The back-annotation process generates a generic timing simulation model. In the Xilinx Development System, NGDAnno back-annotates timing information using an NCD file and, optionally, an NGM file. The NCD file, the output of MAP or PAR, represents the physical design. The NGM file, the output of MAP, represents the logical design. NGDAnno performs either physical back-annotation or logical back-annotation depending on whether an NGM file is supplied as input as follows:

- If you do *not* supply an NGM file, NGDAnno performs physical back-annotation. It produces a simulation model based on the physical implementation only.
- If you supply an NGM file, NGDAnno performs logical back-annotation. It distributes timing information associated with placement, routing, and block configuration from the physical NCD design file onto the logical design represented in the NGM file. If timing delays *cannot* be back-annotated to the logical model, NGDAnno inserts a physical model in place of the logical model. For example, NGDAnno might insert a LUT in place of a group of AND gates. The ARF file shows which instance or net names were replaced. For more information, see the "Hierarchy Changes in Annotated Designs" section.

Note If you make logical changes to an NCD design in the FPGA Editor and change the functional behavior of your design, NGDAnno cannot correlate the changed objects in the physical design with the objects in the logical design. NGDAnno recreates the entire NGA design from the NCD file. A warning indicates that the NCD file is no longer synchronized with the NGM file and that a new NGA file has been created from the NCD file.

For both logical and physical back-annotation, NGDAnno outputs an annotated logical design that has a .nga (Native Generic Annotated) extension. The NGA file is input to the appropriate netlist writer (NGD2EDIF, NGD2VHDL, or NGD2VER). The netlist writer converts the back-annotated file in Xilinx format into netlist format for simulation.

In addition to back-annotating a fully routed design, you can generate simulation netlists at other stages in the design flow. If you want to verify that the logic is correct *before* you map your design, you can use the data in an unmapped NGD file as input to the NGD2EDIF, NGD2VER, or NGD2VHDL program and run a simulation program on the resulting netlist. If you want to simulate with block delays, and not route delays, you can run NGDAnno on the unrouted NCD file generated by the MAP program. Then, run the appropriate netlist writer to generate a simulatable netlist.

Note Block delays are generally 50% of your path delay. Simulating with block delays is an imprecise method of determining whether your timing will be met before you actually place and route.

The following figure shows the back-annotation flow.

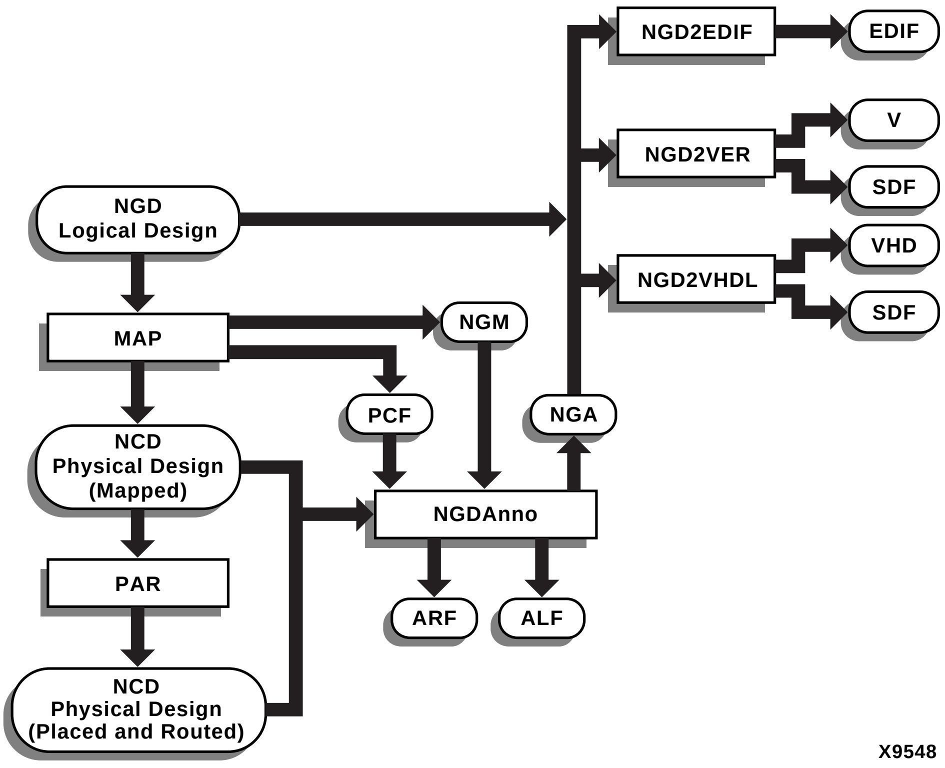


Figure 16-1 Back-Annotation Flow

NGDAnno Syntax

The following command runs NGDAnno:

```
ngdanno [options] ncd_file[.ncd] [ngm_file[.ngm]]
```

ncd_file is the input NCD (physical design file) output from MAP or PAR. If you specify an NCD file on the command line without specifying an NGM file, NGDAnno performs physical back-annotation and the NGA file is generated from the NCD file. The NGA file contains annotated information about the physical implementation only.

ngm_file is an optional NGM file, which is a design file produced by MAP that contains information about the logical design and information about how the logical design corresponds to the physical design. If you specify an NGM file, NGDAnno attempts to annotate physical information onto the logical netlist.

If you do not specify an NGA file with the –o option (described in the "–o (Output File Name)" section), an NGA file is generated in the same directory as the NCD. The NGA file has the same root name as the NCD file. For example, the following command generates an NGA file named mydesign.nga:

```
ngdanno mydesign.ncd
```

NGDAnno Input Files

NGDAnno uses the following files as input:

- NCD file—This physical design file may be mapped only, partially or fully placed, or partially or fully routed.
- NGM file (optional but recommended)—This mapped NGD file is created by the MAP program. This file contains a logical and physical netlist and the correlation of blocks and pins between the two.

The NGM file is useful in the following ways:

- ♦ For schematic-based designs, the NGM file can help regain net names lost in mapping.
- ♦ For HDL synthesis-based designs, the NGM file can help recover the original design hierarchy.

Note When using secure cores, you must use the NGM file as input, or NGDAnno generates an error and does not generate an NGA file.

- PCF file (optional)—This is a physical constraints file.

NGDAnno Output Files

NGDAnno creates the following files as output:

- NGA file—This is a back-annotated design file.

  Note NGA files generated in previous releases cannot be used with the netlist writers (NGD2EDIF, NGD2VHDL, or NGDVER) in this release. You must rerun NGDAnno to generate an NGA file for use with a netlist writer.

- ALF file—This annotation log file contains information about the NGDAnno run, including information on logical annotation failures. The ALF file has the same root name as the output NGA file and an .alf extension. The ALF file is written into the same directory as the output NGA file.

- ARF file—This annotation report file contains information about lost instance or net names. The ARF file has the same root name as the output NGA file and an .arf extension. The ARF file is written into the same directory as the output NGA file.

  This report file is only generated when you use the –report option. See the "–report (Generate Hierarchy Loss Report)" section for more information.

NGDAnno Options

This section describes the NGDAnno command line options.

–f (Execute Commands File)

–f *command_file*

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–module (Simulation of Active Module)

–module *design_name***.ncd**

The –module option creates an NGA file based on the active module, independent of the top-level design. NGDAnno constructs the NGA file based only on the active module's interface signals.

To use this option you must specify an NCD file that contains an expanded active module. To create this NCD file, see the "Implementing an Active Module" section of the "Modular Design" chapter.

For more information on simulating modules, see the "Simulating an Active Module" section of the "Modular Design" chapter.

–o (Output File Name)

–o *out_file*[**.nga**]

The –o option specifies the output design file in NGA format. The .nga extension is optional. The output file name and its location are determined in the following ways:

- If you do not specify an output file name with the –o option, the output file has the same name as the input NCD file, with an .nga extension. The file is placed in the input NCD file's directory.
- If you specify an output file name with no path specifier (for example, **cpu_dec.nga** instead of **/home/designs/cpu_dec.nga**), the NGA file is placed in the current working directory.
- If you specify an output file name with a full path specifier (for example, **/home/designs/cpu_dec.nga**), the output file is placed in the specified directory.

If the output file already exists, it is overwritten with the new NGA file.

–p (PCF File)

-p *pcf_file***.pcf**

The –p option allows you to specify a PCF (Physical Constraints) file as input to NGDAnno. You only need to specify a constraints file if it contains the following:

- Level information (CMOS or TTL) for IOBs in a 4000E or 4000EX design
- Prorating constraints

Prorating constraints and prorated delays are described in the *Constraints Guide.*

–quiet (Report Warnings and Errors Only)

The –quiet option reduces NGDAnno screen output to warnings and errors only. This option is useful if you only want a summary of the NGDAnno run.

–report (Generate Hierarchy Loss Report)

The –report option generates an NGDAnno Report File (ARF). This report contains information about original net and instance names that were lost during back-annotation. Determination of lost nets are based on a flat view of the design. If a net traverses hierarchical boundaries, a loss is reported only if the entire net is removed. This report's introduction explains the reasons for lost net and instance names and provides a key for how to read the report.

This option must be run with an NGM file present. If it is run without this file, NGDAnno issues a warning and does not produce the report file.

Note This option is not recommended for users of synthesis tools that have unpredictable naming conventions.

–s (Change Speed)

```
-s [speed]
```

The –s option instructs NGDAnno to annotate the device speed you specify to the NGA file. The device *speed* can be entered with or without the leading dash. For example, both –s 3 and –s –3 are allowable entries.

Some architectures support the –s min option. This option instructs NGDAnno to annotate a minimum delay, rather than a maximum worst-case delay, to the NGA file. The command line syntax is the following.

```
-s min
```

–s –min is not an allowable entry.

Note Settings made with the –s min option override any prorated timing parameters in the PCF file.

Dedicated Global Signals in Back-Annotation Simulation

This section presents information on how global signals are treated in back-annotation simulation.

Note For a description of the STARTUP, STARTUP_VIRTEX, STARTUP_VIRTEX2, and STARTUP_SPARTAN2 components, see the "Design Elements (SOP3 to XORCY_L)" chapter of the *Libraries Guide*.

XC4000E/L/EX/XL/XLA and Spartan/XL

For the XC4000 device family, Spartan devices, and SpartanXL devices, a High signal on the Global Set/Reset (GSR) net initializes each flip-flop and latch to the state (0 or 1) specified by its INIT property (default is 0). For XC4000 devices, the INIT property must match the flip-flop type, except in the case of FD and LD components. A High signal on Global 3-State (GTS) sets all outputs to a 3-state condition. If you did not use the STARTUP component in your original design, these signals are initialized to their inactive states. Otherwise, you must stimulate the input GSR and GTS pins of the STARTUP device either directly or through logic from explicit pins on the device.

Virtex/-II/-E and Spartan-II

For Virtex, Virtex-II, Virtex-E, and Spartan-II devices, a High signal on the GSR net initializes each flip-flop and latch to the state (0 or 1) specified by its INIT property (default is 0) and each Block RAM data output to 0. The INIT property *must* match the flip-flop type. For example, if you use an FDR flip-flop, you must retain the automatically assigned INIT=R property. The DLL and the contents of the following memory elements are unaffected by GSR: LUT RAM, Block RAM, SRL16, and SRLC16. A High signal on GTS sets all outputs to a 3-state condition. If you did not use the STARTUP_VIRTEX component in your original design, these signals are initialized to their inactive states. Otherwise, you must stimulate the input GSR and GTS pins of the STARTUP_VIRTEX device either directly or through logic from explicit pins on the device.

Virtex-II devices differ slightly from Virtex devices. The startup block for Virtex-II is called STARTUP_VIRTEX2. In addition, GSR has two levels of control. By default when GSR is asserted, a register sets or presets according to its type. For example, an FDR flip-flop changes to 0 when GSR is asserted. With Virtex-II, you can also override this default behavior by using the INIT property. For example, if you assign INIT=S for FDR, asserting GSR changes the state of the register to 1 instead of the default value of 0.

Using a BUFGMUX element in your design may cause inaccurate simulation if all the following conditions occur:

- Both clock inputs (I0 and I1) are used
- GSR is activated during simulation (after simulation time ‘0’)
- The secondary clock input (I1) is selected before or while GSR is active

In this case, the primary clock input (I0) is incorrectly selected. This occurs because there is a cross-coupled register pair that ensures the BUFGMUX output does not inadvertently generate a clock edge. When GSR is asserted, these registers initialize to the default state of I0. To select the secondary clock, you must send a clock pulse to both the primary and secondary clock inputs while GSR is inactive.

Hierarchy Changes in Annotated Designs

If you supply an NGM file as input, NGDAnno attempts to retain your original design hierarchy. However, NGDAnno may flatten part of your original design hierarchy when generating a simulation netlist under the following conditions:

- Logical correlation loss on a CLB, IOB, or slice due to logic optimization and logic replication during mapping
- Failures in logical annotation
- Lost logic is located in different parts of the design hierarchy

For example, if a flip-flop with the hierarchical name A/B/X is merged with a flip-flop named A/C/Y, and the resulting CLB is affected by optimization or logic replication, hierarchical blocks A/B and A/C are flattened out of the netlist. The two flip-flops now lie at the same level of hierarchy (the A level) and are replaced by the CLB physical model.

For information on which blocks were flattened, see the ARF file generated when you use the –report option described in the "–report (Generate Hierarchy Loss Report)" section.

External Setup and Hold Check

In addition to the setup and hold checks already performed during back-annotation, NGDAnno creates an External Setup and Hold Check (ESUH) primitive for *every* input data / clock pair that drives a register. ESUH primitives are created for all Virtex families, XC4000EX devices, and Spartan-II devices. Older device families may not support all ESUH combinations.

ESUH primitives are generated for registers in the design that meet the following requirements:

- Data is available in the speed files.
- Clock and data signals exist at the same hierarchical level.

  Note If a ESUH primitive cannot be created because this requirement is not met, NGDAnno issues a warning.
- The device has an externally driven clock that uses the global clock resources to clock an externally driven register.

Following are details regarding generation of ESUH primitives:

- For data / clock pairs that drive more than one register, NGDAnno uses the maximum setup and maximum hold value of the group.
- NGDAnno does not use timing constraints specified in the PCF file. It only uses the TEMPERATURE, VOLTAGE, and LOGIC constraints specified in the PCF file.

Chapter 17

NGD2EDIF

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes the NGD2EDIF program. The chapter contains the following sections:

NGD2EDIF Overview

NGD2EDIF produces an EDIF 2 0 0 netlist in terms of the Xilinx primitive set, allowing you to simulate pre- and post-route designs. NGD2EDIF can produce an EDIF file representing a design in any of the following stages:

- An unmapped design—To translate an unmapped design, the input to NGD2EDIF is an NGD file—a logical description of your design. The output from NGD2EDIF is an EDIF file containing a functional description of the design without timing information.
- A mapped, unrouted design—To translate a mapped design that has not been placed and routed, the input to NGD2EDIF is an NGA file— an annotated logical description of your design—generated from a mapped physical design. The output from NGD2EDIF is an EDIF file containing a functional description of the design and timing information containing component delays but without routing delays.
- A routed design—To translate a design which has been placed and routed, the input to NGD2EDIF is an NGA file generated from a routed physical design. The output from NGD2EDIF is an EDIF file containing a functional description of the design and timing information containing both component and routing delays.

The following figure shows the design flow for NGD2EDIF.

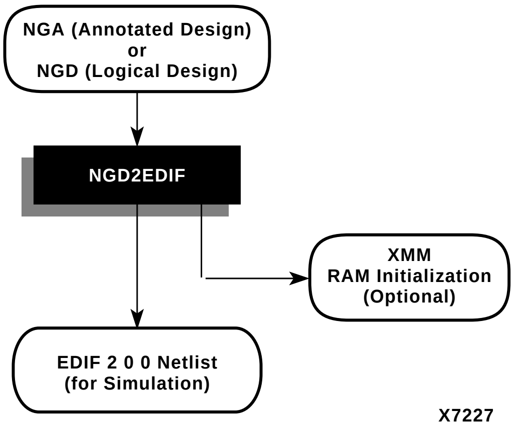


Figure 17-1 NGD2EDIF Design Flow

Note If you use a prohibited core in your design, NGD2EDIF issues an error message and does not export your design. If you use an encrypted core, NGD2EDIF generates an encrypted file.

NGD2EDIF Syntax

The following command invokes the NGD2EDIF translation program:

ngd2edif [*options*] *infile*[**.ngd** | **.nga**] [*outfile*[**.edn**]]

options can be any number of the NGD2EDIF options listed in the "NGD2EDIF Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

infile[**.ngd** | **.nga**] indicates the input file. If you enter a file name without an extension, NGD2EDIF looks for a file with an .nga extension and the name you specified. If you want to translate an NGD file, you must enter the .ngd extension. Without the .ngd extension NGD2EDIF does not use the NGD file as input, even if an NGA file is not present.

outfile[**.edn**] is the name of the NGD2EDIF output file if you want to name it other than the root NGD design name. If you do not give an extension, .edn is added.

NGD2EDIF Input Files

NGD2EDIF uses the following files as input:

- NGA file—This back-annotated logical design file contains Xilinx primitive components.
- NGD file—This logical design file contains Xilinx primitive components.

NGD2EDIF Output Files

NGD2EDIF creates the following files as output:

- EDN file—This is a netlist file in EDIF format. The default EDN file produced by NGD2EDIF is generic. If you want to produce EDIF targeted to Mentor Graphics or Innoveda, you must use the NGD2EDIF –v option (described in the "–v (Vendor)" section).
- XMM file— This optional RAM initialization file defines the initial contents of the RAMs in the design for the simulator. The file is described in the "XMM (RAM Initialization) File" section.

  If an XMM file is generated, it has the same base name and is written into the same directory as the output EDIF netlist.

NGD2EDIF Options

This section describes the NGD2EDIF command line options.

–a (Write All Properties)

The –a option causes NGD2EDIF to write all properties into the output EDIF netlist. The default is to write only timing delay properties and certain other properties that define the behavior of the design logic (for example, RAM INIT properties). In most cases the –a option is not necessary. Check with your simulation vendor on whether this option is a requirement for their tools.

–b (Use Buffers to Model Delays)

The –b option causes NGD2EDIF to model certain delays using buffers. The proper setting for the –b and –i options is chosen automatically if you entered a –v option. If your SimPrim library vendor is not one of the supported values for the –v option, consult the vendor for the proper –b and –i option settings.

–c (Reference Design Name as Specified—Mentor)

The –c option applies to the Mentor Graphics design flow. The option ensures that the output of Mentor's ENRead (EDIF reader) program is an EDDM Single Object simulation model registered to the design component located in the current directory.

If the –c option is *not* specified, a library entry in the EDIF file instructs ENRead to place the simulation model in a subdirectory named *design*_lib. For example, if your design name is adder4, ENRead places the simulation model in the subdirectory adder4_lib/adder4.

If the –c option *is* specified, the library entry in the EDIF file instructs ENRead to place the simulation model directly in the design directory. For example, the simulation model for the design adder4 is placed in the current directory right under adder4 (as opposed to being placed in adder4_lib/adder4). In this directory, the Mentor simulator finds the simulation model.

If you specify the –c option, you must also specify both the –n (Generate Flattened Netlist) option and the –v (Vendor) option (–v mentor).

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter

–hpn (Set HDL Pin Names)

The –hpn option sets SimPrim pin names to HDL compliant names. The pin name IN is set to I, and the pin name OUT is set to O.

–i (Annotate Timing Properties to Instances)

The –i option causes NGD2EDIF to annotate all timing properties to instances. The proper setting for the –i and –b options are chosen automatically if you entered a –v option. If your SimPrim library vendor is not one of the supported values for the –v option, consult the vendor for the proper –i and –b option settings.

–l (Local Scope)

The –l option gives dedicated signals (such as the Global Set/Reset signal) a local (non-global) scope. If your simulation vendor is Mentor Graphics or Innoveda, the default NGD2EDIF action is to give dedicated signals global scope.

If you are simulating a board-level schematic that references more than one Xilinx device, the global dedicated signals from each netlist are implicitly connected by the simulator. If this is not what you want, use the –l option to make the signals local to each device. You then need to reference each dedicated signal by the appropriate hierarchically-qualified signal name.

Note If your simulation vendor is a vendor other than Mentor Graphics or Innoveda, the –l option is enabled by default.

–n (Generate Flattened Netlist)

The –n option writes out a flattened netlist.

–v (Vendor)

```
–v vendor
```

The –v option specifies the CAE vendor toolset that uses the resulting EDIF file. Allowable entries are **viewlog** (for Innoveda) or **mentor** (for Mentor Graphics).

The –v option customizes the output EDIF file for the specified vendor’s simulator. The option also determines whether an XMM (RAM initialization) file is produced and the format of the file (if one is produced). The XMM file is described in the “XMM (RAM Initialization) File” section.

–vpt (Mentor Graphics Viewpoint)

```
–vpt viewpoint_name
```

The –vpt option specifies the desired viewpoint for a Mentor Graphics EDIF output file.

–w (Overwrite Output)

The –w option overwrites the output file.

XMM (RAM Initialization) File

The XMM file defines the initial contents of the RAMs in the design for the simulator. An XMM file is only created if the design contains RAMs. Some simulators require an XMM file; other simulators can read the RAM initialization directly from the output EDIF file and do not need a separate XMM file. Depending on the simulator vendor you specify with the NGD2EDIF –v option, the XMM file is used as follows:

- If you are using an Innoveda simulator (–v viewlog), an XMM file is created in LOADM format for use by ViewSim.
- If you are using a Mentor Graphics simulator (–v mentor), no XMM file is created. QuickSim takes the RAM initialization information directly from the EDIF netlist.
- If you are using another simulator (no –v option), an XMM file is generated in a generic format, which is described in the next section. Your simulator may or may not need this separate file; consult your vendor's documentation for details.

Note RAM initialization data is not created for the Virtex Block RAM.

Generic File Format for XMM File

This section describes the format of the generic XMM file, which is created when the –v option is not specified for NGD2EDIF. Consult your simulator vendor's documentation to determine how to use this generic XMM file.

In most cases you do not need to understand the format of the generic XMM file. The following information is provided for reference. For ease of processing by scripting languages, the generic initialization file consists of newline-separated records. Each record has the following three tokens, separated by white space, with the position of each token denoting its meaning:

```
primitive_type instance_name init_value
```

primitive_type is the name of a RAM primitive in the SimPrim library. It is a string value.

instance_name is a hierarchically-qualified instance name for a particular RAM SimPrim in the design. It is a string value. Hierarchical names are separated by the forward slash (/) character. The *instance_name* is expressed in terms of the names in the original design, not in terms of the EDIF identifiers. The original names are more likely to correlate to the original design, but are not checked for uniqueness and may not be legal for the simulation interface. The simulation interface must read the generic initialization file to resolve these problems.

init_value represents the contents of the specified RAM instance. The *init_value* is a hexadecimal number with a 0x prefix. The most significant bit of this number should be loaded into the highest address of the RAM, continuing so that the least-significant bit is loaded into the lowest address of the RAM. As with the INIT property value, a one-bit-wide RAM is assumed. The number is padded with zeroes so that the number of bits exactly matches the number of addressable locations in the RAM primitive.

Generic Initialization File Example

An example generic initialization file is shown following.

```
X_RAMS16 $1I32/$1I47/FIFO/BANK03 0x6A47
X_RAM32 TOP/IFC/DATA/O7 0x003F097D
X_RAMD16 TOP/$3I107/$7I100 0x0000
```

The generic initialization file also contains several comment lines that document when and how the file was made and describe the file format. Each comment line begins with a pound (#) character; these lines can be ignored by programs using the initialization file.

EDIF Identifier Naming Conventions

An identifier in a EDIF file must adhere to the following conventions:

- Must begin with an alphabetic or ampersand character (a–z, A–Z, or &)
- Can contain alphanumeric (a–z, A–Z, 0-9) or underscore (_) characters

Chapter 18

NGD2VER

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes the NGD2VER program. The chapter contains the following sections:

NGD2VER Overview

NGD2VER translates your design into a Verilog HDL file containing a netlist description of your design in terms of Xilinx simulation primitives. You can use the Verilog file to perform a back-end simulation with a Verilog simulator.

Simulation is based on SimPrims, which create simulation models using basic simulation primitives. For example, because a primitive for the XC4000 dual-port RAM does not exist in the Verilog SimPrim library files, NGD2VER builds a simulation model for the dual-port RAM out of two 16x1 RAM SimPrim primitives.

NGD2VER can produce a Verilog file representing a design at any of the following stages:

- An unmapped design—To translate an unmapped design, the input to NGD2VER is an NGD file—a logical description of your design. The output from NGD2VER is a Verilog file containing a functional description of the design without timing information.
- A mapped, unrouted design—To translate a mapped design which has not been placed and routed, the input to NGD2VER is an NGA file— an annotated logical description of your design— generated from a mapped physical design. The output from NGD2VER is a Verilog file containing a functional description of the design, and an additional SDF (Standard Delay Format) file containing timing information. The SDF file contains component delays without routing delays.
- A routed design—To translate a design that has been placed and routed, the input to NGD2VER is an NGA file generated from a routed physical design. The output from NGD2VER is a Verilog file containing a functional description of the design and an SDF file containing both component and routing delays.

The following figure shows the design flow for NGD2VER.

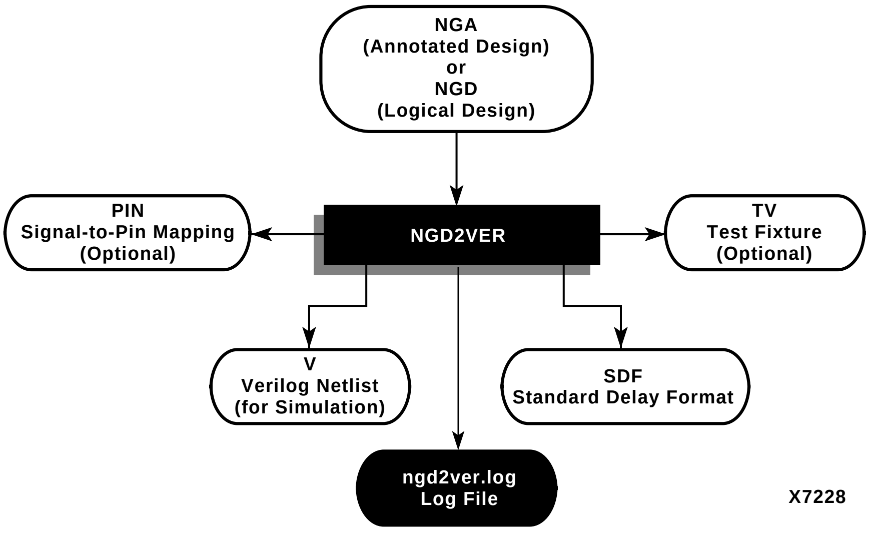


Figure 18-1 NGD2VER Design Flow

Note If you use a prohibited core in your design, NGD2VER issues an error message and does not export your design. If you use an encrypted core, NGD2VER generates an encrypted file.

NGD2VER Syntax

The following command translates your design to a Verilog file:

ngd2ver [*options*] *infile*[**.ngd** | **.nga**] [*outfile*[**.v**]]

options can be any number of the NGD2VER options listed in the "NGD2VER Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

infile [**.ngd** | **.nga**] is the input NGD or NGA file. If you enter a file name with no extension, NGD2VER looks for a file with an .nga extension and the name you specified. If you want to translate an NGD file, you must enter the .ngd extension. Without the .ngd extension NGD2VER does not use the NGD file as input, even if an NGA file is not present.

outfile[.v] indicates the file to which the Verilog output of NGD2VER is written. The default is *infile*.v (*infile* is the same root name as the input file). The SDF file has the same root name as the Verilog file.

NGD2VER Input Files

NGD2VER uses the following files as input:

- NGA—This back-annotated logical design file is produced by NGDAnno and contains Xilinx primitives.
- NGD—This logical design file is produced by NGDBuild and contains Xilinx simulation primitives.

NGD2VER Output Files

NGD2VER creates the following files as output:

- V file—This Verilog HDL file contains the netlist information obtained from the input NGD or NGA file. This file is a simulation model and cannot be synthesized or used in any other manner than simulation. This netlist uses simulation primitives which may not represent the true implementation of the device; however, the netlist represents a functional model of the implemented design. Do not modify this file.
- SDF file—This Standard Delay Format file contains delays obtained from the input file. NGD2VER only generates an SDF file if the input is an NGA file, which contains timing information. The SDF file generated by NGD2VER is based on SDF version 2.1.

  Note The SDF file should only be used with the Verilog file. Do not use the SDF file with the original design or with the product of another netlist writer.
- LOG file—This log file contains all the messages generated during the execution of NGD2VER.
- TV file—This optional test fixture file is created if you use the NGD2VER –tf option.
- PIN file—This is an optional Cadence signal-to-pin mapping file. NGD2VER generates a PIN file if the input file contains routed external pins and you use the NGD2VER –pf option.

NGD2VER only generates an PIN file if the input is an NGA file. The files have the same root name as the NGA file.

NGD2VER Options

This section describes the NGD2VER command line options.

–10ps (Set Time Precision to be 10ps)

The –10ps option changes the default timescale statement from 1 ns / 1 ps to 1 ns / 10 ps. This allows you to choose the appropriate simulation resolution based on your simulation run-time requirements.

–aka (Write Also-Known-As Names as Comments)

The –aka option includes original user-defined identifiers as comments in the Verilog netlist. This option is useful if user-defined identifiers are changed because of name legalization processes in NGD2VER.

–cd (Include `celldefine\`endcelldefine in Verilog File)

The –cd option applies to a Verilog file that will be used with the Cadence Synergy synthesis tool. The –cd option encloses every module definition in `celldefine and `endcelldefine constructs, as in the following example:

```
‘celldefine
  module <module_name>
      .
      .
      .
  endmodule
‘endcelldefine
```

The `celldefine and `endcelldefine constructs instruct the Cadence Synergy software to treat an enclosed module as a black box (that is, do not try to synthesize the enclosed module). Use this option if you use Cadence Synergy and you want to instantiate a LogiBLOX module into the HDL source code.

–f (Execute Commands File)

```
–f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–gp (Bring Out Global Reset Net as Port)

```
-gp port_name
```

The –gp option causes NGD2VER to bring out the Global Reset signal (which is connected to all flip-flops and latches in the physical design) as a port on the top-level module in the output Verilog file. Specifying the port name allows you to match the port name you used in the front-end. The Global Reset signal is discussed in the "Dedicated Global Signals in Back-Annotation Simulation" section of the "NGDAnno" chapter.

This option is only used if the Global Reset net is not driven. For example, if you include a STARTUP component in an XC4000 design, you do not have to enter a –gp option, because the STARTUP component drives the Global Reset net.

Note Do not use GR, GSR, PRELOAD, or RESET as port names, because these are reserved names in the Xilinx software. Also, do not use the name of any wire or port that already exists in the design, because this causes NGD2VER to issue a fatal error.

–ism (Include SimPrim Modules in Verilog File)

The –ism switch includes SimPrim modules from the SimPrim library in the output Verilog (.v) file. This option allows you to bypass specifying the library path during simulation. However, using this switch increases the size of your netlist file and increases your compile time.

When you run this option, NGD2VER checks that your library path is set up properly. Following is an example of the appropriate path:

```
$XILINX/verilog/src/simprim
```

Note If you are using compiled libraries, this switch offers no advantage. If you use this switch, do not use the –ul switch.

–log (Rename the Log File)

```
-log log_file
```

By default, the name of the NGD2VER log file is ngd2ver.log. The –log option allows you to rename the log file. The log file contains all of the messages displayed during the execution of NGD2VER.

–ne (No Name Escaping)

By default (without the –ne option), NGD2VER “escapes” illegal block or net names in your design by placing a leading backslash (\) before the name and appending a space at the end of the name. For example, the net name “p1$40/empty” becomes “\p1$40/empty ” when name escaping is used. Illegal Verilog characters are reserved Verilog names, such as “input” and “output,” and any characters that do not conform to the standards described in the “Verilog Identifier Naming Conventions” section.

The –ne option replaces invalid characters with underscores so that name escaping does not occur. For example, the net name “p1$40/empty” becomes “p1$40_empty” when name escaping is not used. The leading backslash does not appear as part of the identifier. The resulting Verilog file can be used if a vendor’s Verilog software cannot interpret escaped identifiers correctly.

–op (Specify the Period for Oscillator)

```
-op oscillator_period
```

The –op option specifies the period, in nanoseconds, for the oscillator. You must specify a positive integer to stimulate the component properly. If you do not enter a value for the –op option, the default is 100 ns.

Note Use this option only if OSC4 or OSC5 is included in your design.

–pf (Generate Pin File)

The –pf option writes out a pin file—a Cadence signal-to-pin mapping file with a .pin extension.

Note NGD2VER only generates an PIN file if the input is an NGA file.

–pms (Port Names Match Child Signal Names)

The –pms option forces port names and child signal names to match. Ports or pins in the NGD database generally have two connections, one to the port or parent signal and one to the child signal. In most cases, these signal names are the same. If the names are not the same, you can use this option to change the child signal names to match the parent signal name.

–r (Retain Hierarchy)

By default (without the –r option), NGD2VER produces a flattened Verilog HDL file. The –r option writes out a Verilog HDL file that retains the hierarchy in the original design as much as possible. This option groups logic based on the original design hierarchy. To retain the logical hierarchy in your design when using the –r option, you *must* supply an NGM file as input when you run NGDAnno (see the "NGDAnno Input Files" section of the "NGDAnno" chapter). If you do not supply an NGM file, the NGA file produced is based on the physical hierarchy in the NCD file, rather than the original design hierarchy.

Note In some cases, it is not possible to preserve hierarchy. If NGDAnno cannot back-annotate timing delays, it inserts a physical model into the logical model. If the logical elements for any CLB spanned hierarchy, the hierarchy is flattened as a result of this insertion. See the "Hierarchy Changes in Annotated Designs" section of the "NGDAnno" chapter for more information.

–sdf_path (Full Path to SDF File)

```
-sdf_path [path_name]
```

The –sdf_path option outputs the SDF file to the specified full path. This option writes the full path and the SDF file name to the $sdf_annotate statement. If a full path is not specified, it writes the full path of the current work directory and the SDF file name to the $sdf_annotate file.

Note NGD2VER only generates an SDF file if the input is an NGA file, which contains timing information. This option is allowed for an NGA file but not for an NGD file.

–shm (Write $shm Statements in Test Fixture File)

The -shm option places $shm statements in the structural Verilog file created by NGD2VER. These $shm statements allow VerilogXL to display simulation data as waveforms.

–tf (Generate Test Fixture File)

The –tf option generates a test fixture file. The file has a .tv extension, and it is a ready-to-use template test fixture Verilog file based on the input NGD or NGA file.

–ti (Top Instance Name)

```
-ti top_instance_name
```

The –ti option specifies a user instance name for the design under test in the test fixture file created with the -tf option.

–tm (Top Module Name)

```
–tm top_module_name
```

By default (without the –tm option), the output files inherit the top module name from the input NGD or NGA file. The –tm option changes the name of the top-level module name appearing within the NGD2VER output files.

–tp (Bring Out Global 3-State Net as Port)

```
–tp port_name
```

The –tp option causes NGD2VER to bring out the global 3-state signal (which forces all FPGA outputs to the high-impedance state) as a port on the top-level entity in the output Verilog file. Specifying the port name allows you to match the port name you used in the front-end.

This option is only used if the global 3-state net is not driven. For example, if you include a STARTUP component in an XC4000 design, you do not have to enter a –tp option, because the STARTUP component drives the global 3-state net.

Note Do not use the name of any wire or port that already exists in the design, because this causes NGD2VER to issue a fatal error.

–ul (Write ‘uselib Directive)

The –ul option causes NGD2VER to write a library path pointing to the SimPrim library into the output Verilog (.v) file. The path is written as shown following.

```
‘uselib dir=$XILINX/verilog/src/simprims libext=.v
```

$XILINX is the location of the Xilinx software.

If you do not enter a –ul option, the ‘uselib line is not written into the Verilog file.

Note A blank ‘uselib statement is automatically appended to the end of the Verilog file to clear out the ‘uselib data. If you use this option, do not use the –ism option.

–verbose (Report All Messages)

The –verbose option displays detailed Verilog processing messages during the execution of NGD2VER.

–w (Overwrite Existing Files)

The –w option causes NGD2VER to overwrite the .v file if it already exists. By default, NGD2VER does *not* overwrite the .v file.

Note All other output files are automatically overwritten.

Setting Global Set/Reset, 3-State, and PRLD

For information on setting GSR and GTS for FPGAs, see the “Simulating Verilog” section of the *Synthesis and Simulation Design Guide*.

For information on setting Global PRLD for CPLDs, refer to the CPLD Design Techniques online Help.

Test Fixture File

The end of the test fixture (TV) file produced by NGD2VER contains the following commands:

```
#1000 $stop
// #1000 $finish
```

The `$stop` command terminates simulation from the test fixture and places the simulator in “interactive mode.” This mode allows you to view the waveforms produced or allows interaction with other programs that need the simulator open.

To exit automatically instead of entering interactive mode, edit the test fixture file to remove or comment out the $stop line and uncomment the $finish line.

Bus Order in Verilog Files

When you compile your unit-under-test design from NGD2VER along with your test fixture, there may be mismatches on bused ports.

This problem occurs when your unit under test has top-level ports that are defined as LSB-to-MSB, as shown in the following example:

```
input [0:7] A;
```

As a result of the way your input design was converted to a netlist before it was read into the Xilinx implementation software, the Xilinx design database does not include information on how bus direction was defined in the original design. When NGD2VER writes out a structural timing Verilog description, all buses are written as MSB-to-LSB, as shown in the following example:

```
input [7:0] A;
```

If your ports are defined as LSB-to-MSB in your original input design and test fixture, there is a port mismatch when the test fixture is compiled for timing simulation. Use one of the following methods to solve this problem:

- In the test fixture, modify the instantiation of the unit under test so that all ports are defined as MSB-to-LSB for timing simulation.
- Define all ports as MSB-to-LSB in your original design and test fixture. For example, enter **[7:0]** instead of **[0:7]**.

Note Bus order *will* be preserved in the following cases: if the design input file is EDIF and the buses are declared as port arrays, if you are doing logical simulation, or if you are doing back-annotation with an NGM file as input.

Verilog Identifier Naming Conventions

An identifier in a Verilog file must adhere to the following conventions. For more information see the *IEEE Standard Description Language Based on the Verilog™ Hardware Description Language* manual.

- Must begin with an alphabetic or underscore character (a-z, A-Z, or _)
- Can contain alphanumeric (a-z, A-Z, 0-9), underscore (_), or dollar sign ($) characters
- May use any character by escaping with a backslash(\) at the beginning of the identifier and terminating with a white space (a blank, tab, newline, or formfeed). For example, the identifier "reset*" is not acceptable but the identifier "\reset* " is acceptable.
- Can be up to 1024 characters long
- Cannot contain white space

Note Identifiers are case sensitive.

During the name legalization process, NGD2VER writes identifiers that contain invalid characters with a leading backslash and a following white space. If you want to change this default behavior, use the –ne option described in the "–ne (No Name Escaping)" section.

Compile Scripts for Verilog Libraries

You must compile libraries for your simulation tools to recognize Xilinx components. To perform timing or post-synthesis functional HDL simulation, you must compile the SimPrim libraries. If the HDL code contains instantiated components, you must compile the UniSim or LogiBLOX libraries. If the HDL code contains instantiated components from the CORE Generator System, you must compile the CORE Generator behavioral models before you can perform a behavioral simulation. Refer to the *CORE Generator Guide* for more information.

To compile libraries, refer to the "Compiling HDL Libraries" section of the *Synthesis and Simulation Design Guide*.

Note You do not need to compile libraries for Verilog-XL.

Chapter 19

NGD2VHDL

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes the NGD2VHDL program. The chapter contains the following sections:

- "NGD2VHDL Overview"
- "NGD2VHDL Syntax"
- "NGD2VHDL Input Files"
- "NGD2VHDL Output Files"
- "NGD2VHDL Options"
- "VHDL Global Set / Reset Emulation"
- "Bus Order in VHDL Files"
- "VHDL Identifier Naming Conventions"
- "Compile Scripts for VHDL Libraries"

NGD2VHDL Overview

The NGD2VHDL program translates your design into a VHDL file containing a netlist description of the design in terms of Xilinx simulation primitives. You can use the VHDL file to perform a back-end simulation by a VHDL simulator.

Simulation is based on SimPrims, which create simulation models using basic simulation primitives. For example, because a primitive for the XC4000 dual-port RAM does not exist in the VITAL SimPrim library files, NGD2VHDL builds a simulation model for the dual port ram out of two 16x1 RAM SimPrim primitives.

NGD2VHDL produces a VHDL file representing a design in any of the following stages:

- An unmapped design—To translate an unmapped design, the input to NGD2VHDL is an NGD file—a logical description of your design. The output from NGD2VHDL is a VHDL file containing a functional description of the design without timing information.
- A mapped, unrouted design—To translate a mapped design which has not been placed and routed, the input to NGD2VHDL is an NGA file— an annotated logical description of your design—generated from a mapped physical design. The output from NGD2VHDL is a VHDL file containing a functional description of the design, and an additional SDF (Standard Delay Format) file containing timing information. The SDF file contains component delays without routing delays.
- A routed design—To translate a design which has been placed and routed, the input to NGD2VHDL is an NGA file generated from a routed physical design. The output from NGD2VHDL is a VHDL file containing a functional description of the design and an SDF file containing both component and routing delays.

The following figures shows the design flow for NGD2VHDL.

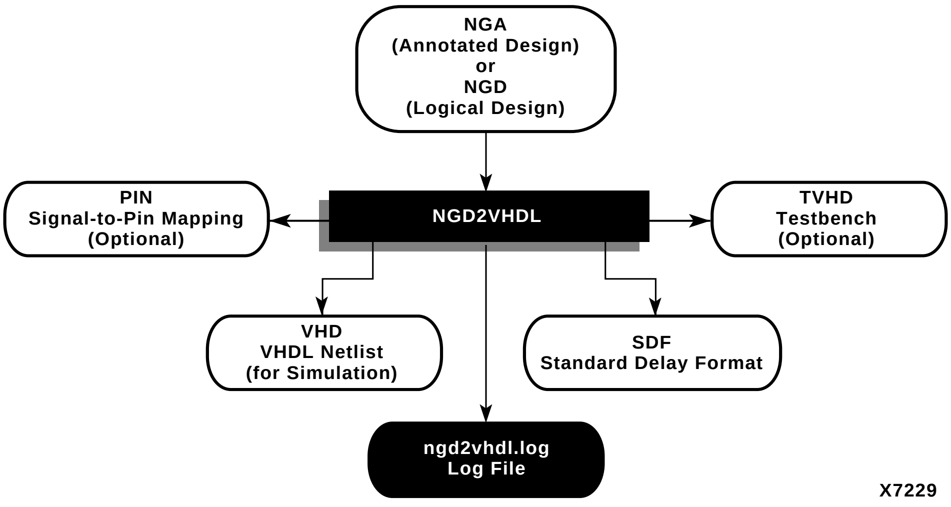


Figure 19-1 NGD2VHDL Design Flow

Note If you use a prohibited core in your design, NGD2VHDL issues an error message and does not export your design. If you use an encrypted core, NGD2VHDL generates an encrypted file.

NGD2VHDL Syntax

The following command translates your design to a VHDL file:

ngd2vhdl [*options*] *infile*[**.ngd** | **.nga**] [*outfile*[**.vhd**]]

options can be any number of the NGD2VHDL options listed in the "NGD2VHDL Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

infile [**.ngd** | **.nga**] is the input NGD or NGA file. If you enter a file name without an extension, NGD2VHDL looks for a file with an .nga extension and the name you specified. If you want to translate an NGD file, you must enter the .ngd extension. Without the .ngd extension NGD2VHDL does not use the NGD file as input, even if an NGA file is not present.

outfile[**.vhd**] indicates the file to which the VHDL output of NGD2VHDL is written. The default is *infile*.vhd (*infile* is the same root name as the input file). The SDF file has the same root name as the VHDL file.

NGD2VHDL Input Files

NGD2VHDL uses the following files as input:

- NGA—This back-annotated logical design file contains Xilinx primitive components.
- NGD—This logical design file contains Xilinx primitive components.

NGD2VHDL Output Files

NGD2VHDL creates the following files as output:

- VHD file—This VITAL 95 and VHDL 87 IEEE compliant VHDL file contains the netlist information obtained from the input NGD or NGA file. This file is a simulation model and cannot be synthesized or used in any other manner than simulation. This netlist uses simulation primitives which may not represent the true implementation of the device; however, the netlist represents a functional model of the implemented design. Do not modify this file.
- SDF file—This Standard Delay Format file contains delays obtained from the input file. NGD2VHDL only generates an SDF file if the input is an NGA file, which contains timing information. The SDF file generated by NGD2VHDL is based on SDF version 2.1.
- LOG file—This log file contains all the messages generated during the execution of NGD2VHDL.
- Testbench file—This optional testbench file is created if you enter the –tb option on the NGD2VHDL command line. The file has a .tvhd extension.
- PIN file—This optional file contains a list of all the input and output pins in the design.

NGD2VHDL Options

This section describes the NGD2VHDL command line options.

–a (Architecture Only)

By default, NGD2VHDL generates both entities and architectures for the input design. If the –a option is specified, no entities are generated and only architectures appear in the output.

–aka (Write Also-Known-As Names as Comments)

The –aka option includes original user-defined identifiers as comments in the VHDL netlist. This option is useful if user-defined identifiers are changed because of name legalization processes in NGD2VHDL.

–ar (Rename Architecture Name)

```
-ar architecture_name
```

The –ar option allows you to rename the architecture name generated by NGD2VHDL. The default architecture name for each entity in the netlist is STRUCTURE.

–f (Execute Commands File)

```
–f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–gp (Bring Out Global Reset Net as Port)

```
–gp port_name
```

The –gp option causes NGD2VHDL to bring out the Global Reset signal (which is connected to all flip-flops and latches in the physical design) as a port on the top-level entity in the output VHDL file. Specifying the port name allows you to match the port name you used in the front-end. The Global Reset signal is discussed in the "VHDL Global Set/Reset Emulation" section.

This option is only used if the Global Reset net is not driven. For example, if you include a STARTUP component in an XC4000 design, you do not have to enter a –gp option, because the STARTUP component drives the Global Reset net.

Note Do not use GR, GSR, PRELOAD, or RESET as port names, because these are reserved names in the Xilinx software.

–log (Specify the Log File)

–log *log_file*

By default, the name of the NGD2VHDL log file is ngd2vhdl.log. The –log option allows you to rename the log file. The log file contains all of the messages displayed during the execution of NGD2VHDL.

–op (Specify the Period for Oscillator)

–op *oscillator_period*

The –op option specifies the period, in nanoseconds, for the oscillator. You must specify a positive integer to stimulate the component properly. If you do not enter a value for the –op option, the default is 100 ns.

Note Use this option only if OSC4 or OSC5 is included in your design.

–pf (Generate PIN File)

The –pf option generates a PIN file that contains a list of all the input and output pins in the design.

–pms (Port Names Match Child Signal Names)

The –pms option forces port names and child signal names to match. Ports or pins in the NGD database generally have two connections, one to the port or parent signal and one to the child signal. In most cases, these signal names are the same. If the names are not the same, you can use this option to change the child signal names to match the parent signal name.

–r (Retain Hierarchy)

By default (without the –r option), NGD2VHDL produces a flattened VHDL file. The –r option writes out a VHDL file that retains the hierarchy in the original design as much as possible. This option groups logic based on the original design hierarchy. To retain the logical hierarchy in your design when using the –r option, you *must* supply an NGM file as input when you run NGDAnno (see the "NGDAnno Input Files" section of the "NGDAnno" chapter). If you do not supply an NGM file, the NGA file produced is based on the physical hierarchy in the NCD file, rather than the original design hierarchy.

Note In some cases, it is not possible to preserve hierarchy. If NGDAnno cannot back-annotate timing delays, it inserts a physical model into the logical model. If the logical elements for any CLB spanned hierarchy, the hierarchy is flattened as a result of this insertion. See the "Hierarchy Changes in Annotated Designs" section of the "NGDAnno" chapter for more information.

–rpw (Specify the Pulse Width for ROC)

–rpw *roc_pulse_width*

The –rpw option specifies the pulse width, in nanoseconds, for the ROC component. You must specify a positive integer to stimulate the component properly. This option is not required. By default, the ROC pulse width is set to 100 ns.

–tb (Generate Testbench File)

The –tb option writes out a testbench file with a .tvhd extension. The default top-level instance name within the testbench file is UUT. If you enter a –ti (Top Instance Name) option, the top-level instance name is the name specified by the –ti option.

–te (Top Entity Name)

–te *top_entity_name*

By default (without the –te option), the output files inherit the top entity name from the input NGD or NGA file. The –te option specifies the name of the top-level entity in the structural VHDL file produced by NGD2VHDL for timing simulation.

–ti (Top Instance Name)

`–ti` *top_instance_name*

The –ti option specifies the name of the top-level instance name appearing within the output SDF file and testbench file (if produced).

The option allows you to match the top-level instance name to the name specified in your test driver VHDL file. Without this option, the SDF file generated by NGD2VHDL cannot be processed properly by VHDL simulators (for example, Model Technology vsim) for timing simulation.

If you do not enter a –ti option, the output files contain a top-level instance name of UUT.

–tp (Bring Out Global 3-State Net as Port)

`–tp` *port_name*

The –tp option causes NGD2VHDL to bring out the global 3-state signal (which forces all FPGA outputs to the high-impedance state) as a port on the top-level entity in the output VHDL file. Specifying the port name allows you to match the port name you used in the front-end.

This option is only used if the global 3-state net is not driven. For example, if you include a STARTUP component in an XC4000 design, you do not have to enter a –tp option, because the STARTUP component drives the global 3-state net.

–tpw (Specify the Pulse Width for TOC)

`–tpw` *toc_pulse_width*

The –tpw option specifies the pulse width, in nanoseconds, for the TOC component. You must specify a positive integer to stimulate the component properly. This option is required when you instantiate the TOC component (for example, when the Global Set/Reset and Global 3-State nets are sourceless in the design).

–verbose (Report All Messages)

The –verbose option displays detailed VHDL processing messages when NGD2VHDL is run.

–w (Overwrite Existing Files)

The –w option causes NGD2VHDL to overwrite the .vhd file if it exists. By default, NGD2VHDL does *not* overwrite the .vhd file.

Note All other output files are automatically overwritten.

–xon (Select Output Behavior for Timing Violations)

```
–xon {true|false}
```

The –xon option specifies the output behavior when timing violations occur on memory elements. If you set this option to true, any memory elements that violate a setup time trigger X on the outputs. If you set this option to false, the signal’s previous value is retained. If you do not set this option, –xon true is run.

VHDL Global Set/Reset Emulation

VHDL requires ports for all signals to be controlled by a testbench. There are VHDL specific components that can be instantiated in the RTL and post-synthesis VHDL description in order to enable the simulation of the global signals for Global Set / Reset (GSR) and Global 3-State (GTS). NGD2VHDL creates a port on the back-annotated design entity for stimulating the GSR or GTS enable signal. This port does not actually exist on the configured part.

You do not need to use the –gp switch to create an external port if you instantiated a STARTUP block in the implemented design. In this case, the port is already identified and connected to the GSR or GTS enable signal. If you do not use the –gp option or a STARTUP block, you will need to use a special cell. Detailed directions for specific emulation cells and their uses follow.

Note The term “STARTUP” refers to the STARTUP block for all device families, including the Virtex STARTUP block, STARTUP_VIRTEX, and the Virtex-II STARTUP block, STARTUP_VIRTEX2. The term “STARTBUF” refers to the STARTBUF cell for all device families, including the Virtex STARTBUF cell, STARTBUF_VIRTEX, and the Virtex-II STARTBUF cell, STARTBUF_VIRTEX2.

VHDL Only STARTUP Block

The STARTUP block is traditionally instantiated to identify the GR, PRLD, or GSR signals for implementation. However, the only time simulation is enabled in the traditional method is when the net attached to the GSR or GTS also goes off chip, because the STARTUP block does not have simulation models. You can use the following new cells to simulate global set/reset or 3-state nets in all cases, whether or not the signal goes off chip.

Note The Virtex and Virtex-II STARTUP blocks are subsets of the XC4000 STARTUP block. However, they differ from the XC4000 STARTUP block in that they have no outputs, as shown in the following figure.

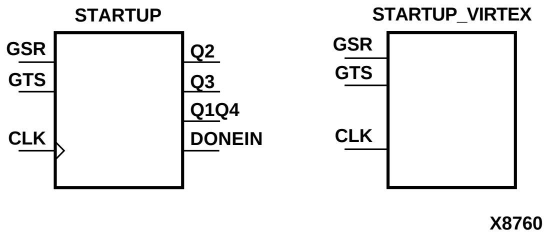


Figure 19-2 STARTUP and STARTUP_VIRTEX Blocks

VHDL Only STARTBUF Cell

The STARTBUF cell passes a Reset or 3-State signal in the same way that a buffer allows simulation to proceed, and it also instantiates the STARTUP block for implementation. STARTBUF is a more simulation friendly version of a typical STARTUP block. Implementation with the correct STARTUP block is handled automatically. Following is an instantiation example for the STARTBUF cell:

```
U1: STARTBUF port map (GSRIN => DEV_GSR_PORT, GTSIN
=>DEV_GTS_PORT, CLKIN => ‘0’, GSROUT => GSR_NET,
GTSOUT => GTS_NET, Q2OUT => open, Q3OUT => open,
Q1Q4OUT => open, DONEINOUT => open):
```

One or both of the input ports GSRIN and GTSIN of the STARTBUF component and the associated output ports GSROUT and GTSOUT can be used. The pins that are left “open” can be used to pass configuration instructions down to implementation, just as on a traditional STARTUP block. You can do this by connecting the appropriate signal to the port instead of leaving it in an "open" condition.

Note The STARTBUF_VIRTEX and STARTBUF_VIRTEX2 cells are similar to the STARTBUF cell, but GSROUT is not available.

VHDL Only STARTUP_VIRTEX Block and STARTBUF_VIRTEX Cell

Global Set/Reset and Global 3-State for the Virtex STARTUP block, STARTUP_VIRTEX, and STARTBUF cell, STARTBUF_VIRTEX, operate as described in the preceding sections with the following qualifications:

- Pre-NGDBuild UniSim VHDL simulation of the GSR signal is not supported.

  The simulation libraries will start up in the correct state; however, you cannot reset the design after simulation time '0.'
- During Pre-NGDBuild UniSim VHDL simulation, designs are properly initialized at simulation time '0.'
- Post-NGDBuild SimPrim VHDL simulation of GSR is supported.

  To correctly back-annotate a GSR signal, instantiate a STARTUP_VIRTEX or STARTBUF_VIRTEX symbol and correctly connect the GSR input signal of that component. When back-annotated, your GSR signal is correctly connected to the associated registers and RAM blocks.
- Pre-NGDBuild UniSim VHDL simulation of the GTS signal is supported.

  Instantiate either a STARTBUF_VIRTEX, TOC, or TOCBUF for this functionality.

Note This information also applies to STARTUP_VIRTEX2 and STARTBUF_VIRTEX2.

VHDL Only RESET-ON-CONFIGURATION (ROC) Cell

This cell is created during back-annotation if you do not use the –gp option or STARTUP block options. It can be instantiated in the front end to match functionality with GSR, GR, or PRLD. (This is done in both functional and timing simulation.) During back-annotation, the entity and architecture for the ROC cell is placed in the design's output VHDL file. In the front end, the entity and architecture are in the UniSim Library, and require only a component instantiation.

The ROC cell generates a one-time initial pulse to drive the GR, GSR, or PRLD net starting at time '0' for a user-defined pulse width. You can set the pulse width with a generic in a configuration statement. The default value of "width" is 0 ns, which disables the ROC cell and results in the Global Set/Reset being held Low. (Active-Low resets are handled within the netlist itself and require you to invert this signal before using it.)

The ROC cell allows you to simulate with the same testbench as in the RTL simulation, and also allows you to control the width of the global set/reset signal in the implemented design.

The ROC components require a value for the generic WIDTH, usually specified with a configuration statement. Otherwise, a generic map is required as part of the component instantiation.

You can set the generic with any generic mapping method you choose. Set the "width" generic after consulting *The Programmable Logic Data Book* for the particular part and mode you have implemented.

For example, a XC4000E part can vary from 10 ms to 130 ms. The value to look for is the TPOR (Power-ON Reset) parameter found in the Configuration Switching Characteristics tables for master, slave, and peripheral modes.

One of the easiest methods for mapping the generic is a configuration for the user's testbench. Following is an example testbench configuration for setting the generic:

```
CONFIGURATION cfg_my_timing_testbench OF my_testbench IS
  FOR my_testbench_architecture
    FOR ALL:my_design USE ENTITY work.my_design(structure);
      FOR structure
        FOR ALL:roc ENTITY USE work.roc (roc_v)
          Generic MAP (width => 100 ms);
        END FOR;
      END FOR;
    END FOR;
  END FOR;
END cfg_my_timing_testbench;
```

The following is an instantiation example for the ROC cell.

```
U1: ROC port map (0 =>GSR_NET);
```

VHDL Only ROCBUF Cell

The ROCBUF allows you to provide stimulus for the Reset on Configuration signal through a testbench but the port connected to it is not implemented as a chip pin. The port can be brought back in the timing simulation with the –gp switch on NGD2VHDL. Following is an example of instantiation of the ROCBUF cell:

```
U1: ROCBUF port map (I => SIM_GSR_PORT, O => GSR_NET);
```

Note This cell is not available for Virtex, Virtex-E, Virtex-II, and Spartan-II devices.

VHDL Only 3-State-On-Configuration (TOC) Cell

This cell is created if you do not use the –tp or STARTUP block options. The entity and architecture for the TOC cell is placed in the design's output VHDL file. The TOC cell generates a one-time initial pulse to drive the GR, GSR, or PRLD net starting at time '0' for a user-defined pulse width. The pulse width can be set with a generic. The default value of "width" is 0 ns, which disables the TOC cell and results in the 3-State enable being held Low. (Active-Low 3-State enables are handled within the netlist itself and require you to invert this signal before using it.)

The TOC cell allows you to simulate with the same testbench as in the RTL simulation, and also allows you to control the width of the 3-State enable signal in the implemented design.

The TOC components require a value for the generic WIDTH, usually specified with a configuration statement. Otherwise, a generic map is required as part of the component instantiation.

You may set the generic with any generic mapping method you choose. Set the “width” generic after consulting *The Programmable Logic Data Book* for the particular part and mode you have implemented.

For example, a XC4000E part can vary from 10 ms to 130 ms. The value to look for is the TPOR (Power-ON Reset) parameter found in the Configuration Switching Characteristics tables for master, slave, and peripheral modes.

One of the easiest methods for mapping the generic is a configuration for the user’s testbench. Following is an example testbench configuration for setting the generic:

```
CONFIGURATION cfg_my_timing_testbench OF my_testbench IS
  FOR my_testbench_architecture
    FOR ALL:my_design USE ENTITY work.my_design(structure);
      FOR structure
        FOR ALL:toc ENTITY USE work.toc (toc_v)
          Generic MAP (width => 100 ms);
        END FOR;
      END FOR;
    END FOR;
  END FOR;
END cfg_my_timing_testbench;
```

VHDL Only TOCBUF

The TOCBUF allows you to provide stimulus for the Global 3-State signal through a testbench but the port connected to it is not implemented as a chip pin. The port can be brought back in the timing simulation with the –tp switch on NGD2VHDL. Following is an example of the instantiation of the TOCBUF cell:

```
U2: TOCBUF port map (I =>SIM_GTS_PORT, O =>GTS_NET);
```

VHDL Only Oscillators

Oscillator output can vary within a fixed range. The cell is not included in the SimPrim library, because you cannot drive global signals in VHDL designs. Schematic simulators can define and drive global nets so the cell is not required. Verilog has the ability to drive nets within a lower level module as well. Therefore the oscillator cells are only required in VHDL. After back-annotation, their entity and architectures are contained in the design's VHDL output.

For functional simulation, they may be instantiated and simulated with the UniSim Library.

The period of the base frequency must be set in order for the simulation to proceed, because the default period of 0 ns disables the oscillator. The oscillator's frequency can very significantly with process and temperature.

Before you set the base period parameter, consult *The Programmable Logic Data Book* for the particular part you are using. For example, the section in *The Programmable Logic Data Book* for the XC4000 Series On-Chip Oscillator states that the base frequency can vary from 4 MHz to 10 MHz, and is nominally 8 MHz. This means the base period generic "period_8m" in the XC4000E OSC4 VHDL model can range from 250 ns to 100 ns. Example of this follow.

Note The OSC4 cell is only available for the XC4000 device family and for Spartan and SpartanXL devices.

Example 1: Oscillator VHDL

Following is an example of the XC4000E OSC4 VHDL model.

```
library IEEE;

use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library UNISIM;
use UNISIM.all;

entity test1 is
port (DATAIN: in STD_LOGIC;
DATAOUT: out STD_LOGIC);
end test1;
```

```
architecture inside of test1 is

signal RST: STD_LOGIC;

component ROC
port(O: out STD_LOGIC);
end component;

component OSC4
port(F8M: out STD_LOGIC);
end component;

signal internalclock: STD_LOGIC;
begin

U0: ROC port map (RST);

U1: OSC4 port map (F8M=>internalclock);

process(internalclock)
begin
if (RST='1') then
DATAOUT <= '0';

elsif(internalclock'event and internalclock='1') then
DATAOUT <= DATAIN;

end if;

end process;

end inside;
```

Example 2: Oscillator Test Bench

Following is a second example.

```
library IEEE;

use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
library UNISIM;
use UNISIM.all;

entity test_oftest1 is end test_oftest1;

architecture inside of test_oftest1 is

component test1
port(DATAIN: in STD_LOGIC;
DATAOUT: out STD_LOGIC);
end component;

signal userdata, userout: STD_LOGIC;

begin

UUT: test1 port map(DATAIN=>userdata,DATAOUT=>userout);

myinput: process
begin
userdata <= ’1’;
wait for 299 ns;
userdata <= ’0’;wait for 501 ns;
end process;

end inside;

configuration overall of test_oftest1 is
for inside
  for UUT:test1
    for inside
      for U0:ROC use entity UNISIM.ROC(ROC_V)
      generic map (WIDTH=> 52 ns);
      end for;

      for U1:OSC4 use entity UNISIM.OSC4(OSC4_V)
      generic map (PERIOD_8M=> 25 ns);
      end for;
    end for;
  end for;
end for;
end overall;
```

This configuration is for pre-NGDBuild simulation. A similar configuration is used for post-NGDBuild simulation. The ROC, TOC, and OSC4 are mapped to the WORK library, and corresponding architecture names may be different. Review the .vhd file created by NGD2VHDL for the current entity and architecture names for post-NGDBuild simulation.

Bus Order in VHDL Files

When you compile your unit-under-test design from NGD2VHDL with your testbench, there may be mismatches on bused ports.

This problem occurs when your unit under test has top-level ports that are defined as LSB-to-MSB, as shown in the following example:

```
A: in STD_LOGIC_VECTOR (0 to 7);
```

As a result of the way your input design was converted to a netlist before it was read into the Xilinx implementation software, the Xilinx design database does not include information on how bus direction was defined in the original design. When NGD2VHDL writes out a structural timing VHDL description, all buses are written as MSB-to-LSB, as shown in the following example:

```
A: in STD_LOGIC_VECTOR (7 downto 0);
```

If your ports were defined as LSB-to-MSB in your original input design and testbench, there is a port mismatch when the testbench is compiled for timing simulation. Use one of the following methods to solve this problem:

- In the testbench, modify the instantiation of the unit under test so that all ports are defined as MSB-to-LSB for timing simulation.
- Define all ports as MSB-to-LSB in the original design and testbench, by using the downto clause instead of the to clause to specify a bus range.

Note Bus order *will* be preserved in the following cases: if the design input file is EDIF and the buses are declared as port arrays, if you are doing logical simulation, or if you are doing back-annotation with an NGM file as input.

VHDL Identifier Naming Conventions

An identifier in a VHDL file must adhere to the following conventions. For more information see the *IEEE Standard VHDL Language Reference Manual or the IEEE Standard VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*.

- Must begin with alphabetic characters (a–z or A–Z)
- Can contain alphanumeric (a–z, A–Z, 0-9) or underscore (_) characters
- Can be up to 1024 characters long
- Cannot contain white space

Note Identifiers are *not* case sensitive.

During the name legalization process, NGD2VHDL substitutes any illegal characters with the underscore (_) character.

Compile Scripts for VHDL Libraries

You must compile libraries for your simulation tools to recognize Xilinx components. To perform timing or post-synthesis functional HDL simulation, you must compile the SimPrim libraries. If the HDL code contains instantiated components, you must compile the UniSim or LogiBLOX libraries. If the HDL code contains instantiated components from the CORE Generator System, you must compile the COREGen behavioral models before you can perform a behavioral simulation. Refer to the *CORE Generator User Guide* for more information.

To compile libraries, refer to the “Compiling HDL Libraries” section of the *Synthesis and Simulation Design Guide*.

Chapter 20

XFLOW

XFLOW is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

This chapter describes the XFLOW program, a scripting tool which allows you to automate implementation and simulation flows using Xilinx programs. It contains the following sections:

XFLOW Overview

XFLOW is a command line tool that allows you to automate the Xilinx implementation and simulation flows. XFLOW reads a design file as input as well as a flow file and option files. When you specify a flow type (described in the "XFLOW Flow Types" section), XFLOW calls a particular flow file. Xilinx provides a default set of flow files that specify which Xilinx programs to run to achieve a certain flow. For example, a flow file could specify that NGDBuild, MAP, PAR, and TRACE should be run to achieve an implementation flow for an FPGA. You can use the default set of flow files as is, or you can modify them. See the "Flow Files" section for more information. Option files specify which command line options should be run for each of the programs listed in the flow file. You can use the default set of option files provided by Xilinx, or you can create your own option files. See the "XFLOW Options" section for more information.

The following figure shows the inputs and the possible outputs of the XFLOW program. The output files depend on the flow you run.

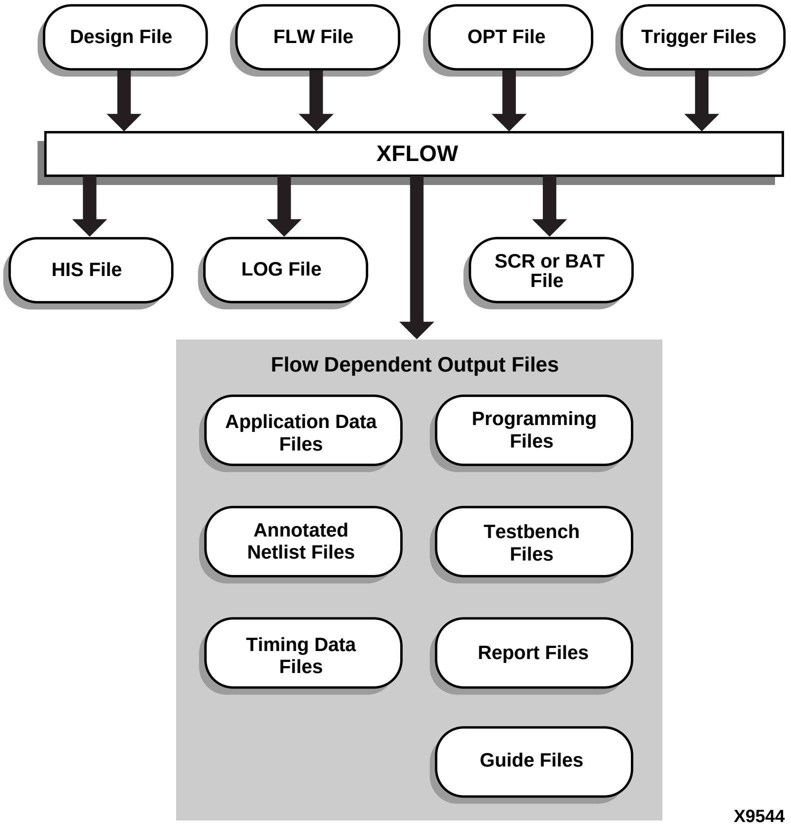


Figure 20-1 XFLOW Design Flow

XFLOW Syntax

Following is the syntax for XFLOW:

```
xflow [flow type] [option file[.opt]] [xflow option]
design_name
```

flow type can be any of the flow types listed in the "XFLOW Flow Types" section. Specifying a flow type instructs XFLOW to read a certain flow file. You can combine multiple flow types on one command line, but each flow type must have its own option file.

option file can be any of the option files that are valid for the specified flow type. See the "XFLOW Option Files" section for more information. In addition, option files are described in the applicable flow type section.

xflow options can be any of the options described in the "XFLOW Options" section. They can be listed in any order. Separate multiple options with spaces.

design_name is the name of the top-level design file you want to process. See the "XFLOW Input Files" section for a description of input design file formats.

Note If you specify a design name only and do not specify a flow type or option file, XFLOW defaults to the –implement flow type and fast_runtime.opt option file for FPGAs and the –fit flow type and balanced.opt option file for CPLDs.

You do not need to specify the complete path for option files. By default, XFLOW uses the option files in your working directory. If the option files are not in your working directory, XFLOW searches for them in the following locations and copies them to your working directory. If XFLOW cannot find the option file in any of these locations, it issues an error message.

- Directories specified using XIL_XFLOW_PATH
- Installed area specified with the XILINX environment variable

Note By default, the directory from which you invoked XFLOW is your working directory. If you want to specify a different directory, use the –wd option described in the "–wd (Specify a Working Directory)" section.

XFLOW Input Files

XFLOW uses the following files as input:

- Design File (for non-synthesis flows)—For all flow types except –synth, the input design can be an XNF, EDIF 2 0 0, PLD, or NGC (XST output) netlist file. You can also specify an NGD, NGO, or NCD file if you want to start at an intermediate point in the flow. XFLOW recognizes and processes files with the extensions shown in the following table.

File Type	Recognized Extensions
EDIF	.sedif, .edn, .edf, .edif
NCD	.ncd
NGC	.ngc
NGD	.ngd
NGO	.ngo
PLD	.pld
XNF	.xtf, .xg, .xff, .sxnf, .xnf

- Design File (for synthesis flows)—For the –synth flow type, the input design can be a Verilog or VHDL file. If you have multiple VHDL or Verilog files, you can use a PRJ or V file that references these files as input to XFLOW. For information on creating a PRJ or V file, see the "Example 1: How to Synthesize VHDL Designs Using Command Line Mode" section or the "Example 2: How to Synthesize Verilog Designs Using Command Line Mode" section of the *Xilinx Synthesis Technology (XST) User Guide*. You can also use existing PRJ files generated while using the Project Navigator. XFLOW recognizes and processes files with the extensions shown in the following table.

File Type	Recognized Extensions
EDIF	.sedif, .edn, .edf, .edif
PRJ	.prj
Verilog	.v, .vf
VHDL	.vhd, .vhf

- FLW File—The flow file is an ASCII file that contains the information necessary for XFLOW to run an implementation or simulation flow. When you specify a flow type (described in the "XFLOW Flow Types" section), XFLOW calls a particular flow file. The flow file contains a program block for each program invoked in the flow. It also specifies the directories in which to copy the output files. You can use the default set of flow files as is, or you can modify them. See the "Flow Files" section for more information.
- OPT Files—Option files are ASCII files that contain options for each program included in a flow file. You can create your own option files or use the ones provided by Xilinx. See the "XFLOW Option Files" section for more information.
- Trigger Files—Trigger files are any additional files that a command line program reads as input, for example, UCF, NCF, PCF, and MFP files. Instead of specifying these files on the command line, these files must be listed in the Triggers line of the flow file. See the "Flow File Format" section for more information.

XFLOW Output Files

XFLOW always outputs the following files. These files are written to your working directory.

- HIS file—The xflow.his file is an ASCII file that contains the XFLOW command you entered to execute the flow, the flow and option files used, the command line commands of programs that were run, and a list of input files for each program in the flow.
- LOG file—The xflow.log file is an ASCII file that contains all the messages generated during the execution of XFLOW.
- SCR or BAT file—This script file contains the command line commands of all the programs run in a flow. This file is created for your convenience, in case you want to review all the commands run, or if you want to execute the script file at a later time. The file extension varies depending on your platform.

In addition, XFLOW outputs one or more of the files shown in the following tables. The output files generated depend on the programs included in the flow files and the commands included in the option files.

Note Report files are written to the working directory by default. You can specify a different directory by using the XFLOW –rd option, described in the "–rd (Copy Report Files)" section, or by using the Report Directory option in the flow file, described in the "Flow Files" section. All report files are in ASCII format.

The following table lists files that can be generated for both FPGAs and CPLDs.

Table 20-1 XFLOW Output Files (FPGAs and CPLDs)

File Name	Description	To Generate this File...
design_name.bld	This report file contains information about the NGDBuild run, in which the input netlist is translated to an NGD file.	Flow file must include "ngdbuild" (Use the –implement or –fit flow type)
time_sim.edn func_sim.edn	This EDIF netlist is a simulation netlist expressed in terms of Xilinx simulation primitives. It differs from the EDIF input netlist and should only be used for simulation, not implementation.	Flow file must include "ngd2edif" (Use the –tsim or –fsim flow type)
time_sim.sdf func_sim.sdf	This Standard Delay Format file contains the timing data for a design.	Flow file must include "ngd2ver" or "ngd2vhdl" (Use the –tsim or –fsim flow type) Input must be an NGA file, which includes timing information.
time_sim.tv func_sim.tv	This is an optional Verilog test fixture file.	Flow file must include "ngd2ver" (Use the –tsim or –fsim flow type) Option file must include NGD2VER –tf option

Table 20-1 XFLOW Output Files (FPGAs and CPLDs)

File Name	Description	To Generate this File...
time_sim.tvhd func_sim.tvhd	This is an optional VHDL test-bench file.	Flow file must include “ngd2vhdl” (Use the –tsim or –fsim flow type) Option file must include NGD2VHDL –tb option
time_sim.v func_sim.v	This Verilog netlist is a simulation netlist expressed in terms of Xilinx simulation primitives. It differs from the Verilog input netlist and should only be used for simulation, not implementation.	Flow file must include “ngd2ver” (Use the –tsim or –fsim flow type)
time_sim.vhd func_sim.vhd	This VHDL netlist is a simulation netlist expressed in terms of Xilinx simulation primitives. It differs from the VHDL input netlist and should only be used for simulation, not implementation.	Flow file must include “ngd2vhdl” (Use the –tsim or –fsim flow type)

The following table lists the output files that can be generated for FPGAs.

Table 20-2 XFLOW Output Files (FPGAs)

File Name	Description	To Generate this File...
design_name.bgn	This report file contains information about the BitGen run, in which a bitstream is generated for Xilinx device configuration.	Flow file must include “bitgen” (Use the –configuration flow type)
design_name.bit	This bitstream file contains configuration data that can be downloaded to an FPGA using the Prom File Formatter, PromGen, or iMPACT.	Flow file must include “bitgen” (Use the –configuration flow type)
design_name.dly	This report file lists delay information for each net in a design.	Flow file must include “par” (Use the –implement flow type)
design_name.ll	This optional ASCII file describes the position of latches, flip-flops, and IOB inputs and outputs in the BIT file.	Flow file must include “bitgen” (Use the –configuration flow type) Option file must include BitGen –l option
design_name.mrp	This report file contains information about the MAP run, in which a logical design is mapped to a Xilinx FPGA.	Flow file must include “map” (Use the –implement flow type)
design_name.ncd	This Native Circuit Description file can be used as a guide file. It is a physical description of the design in terms of the components in the target Xilinx device. This file can be a mapped NCD file or a placed and routed NCD file.	Flow file must include “map” or “par” (Use the –implement flow type)
design_name.par	This report file contains summary information of all placement and routing iterations.	Flow file must include “par” (Use the –implement flow type)

Table 20-2 XFLOW Output Files (FPGAs)

File Name	Description	To Generate this File...
design_name.pad	This report file lists all I/O components used in the design and their associated primary pins.	Flow file must include "par" (Use the –implement flow type)
design_name.rbt	This optional ASCII "rawbits" file contains ones and zeros representing the data in the bitstream file.	Flow file must include "bitgen" (Use the –configuration flow type) Option file must include BitGen –b option
design_name.twr	This report file contains timing data calculated from the NCD file.	Flow file must include "trce" (Use the –implement flow type)
time_sim.xmm	This memory initialization file defines the initial contents of the RAMs in the design for a simulator.	Device family must be one of the following: XC4000E/EX Flow file must include "ngd2edif" (Use the –tsim or –fsim flow type)
design_name.xpi	This report file contains information on whether the design routed and timing specifications were met.	Flow file must include "par" (Use the –implement flow type)

The following table lists the output files that can be generated for CPLDs.

Table 20-3 XFLOW Output Files (CPLDs)

File Name	Description	To Generate this File...
design_name.gyd	This ASCII file is a CPLD guide file.	Flow file must include “cpldfit” (Use the –fit flow type)
design_name.jed	This ASCII file contains configuration data that can be downloaded to a CPLD using iMPACT.	Flow file must include “hprep6” (Use the –fit flow type)
design_name.rpt	This report file contains information about the CPLD Fitter run, in which a logical design is fit to a CPLD.	Flow file must include “cpldfit” (Use the –fit flow type)
design_name.tim	This report file contains timing data.	Flow file must include “taengine” (Use the –fit flow type)

Note If you are using a VM6 file as input to generate a JED, TIM, or NGA file, you must create a new VM6 file using the current CPLD Fitter. VM6 files generated in previous releases of the Xilinx software cannot be used.

XFLOW Flow Types

A “flow” is a sequence of programs invoked to implement, configure, or simulate a design. For example, to implement an FPGA design, the design is run through the NGDBuild, MAP, and PAR program flow.

“Flow types” are XFLOW command line commands that instruct XFLOW to execute a particular flow as specified in a flow file. (For more information on flow files, including how you can create your own, see the “Flow Files” section.) You can enter multiple flow types on the command line to achieve a desired flow. Following are the flow types you can use.

Note All flow types require that an option file be specified. If you do not specify an option file, XFLOW issues an error.

–assemble (Module Assembly)

```
–assemble option_file –pd pim_directory_path
```

Note This flow type is supported for Virtex / -II / -II PRO / -E and Spartan-II / -IIE device families only.

This flow type runs the final phase of the Modular Design flow. In this “Final Assembly” phase, the team leader assembles the top-level design and modules into one NGD file and then implements this file.

This flow type invokes the fpga.flw flow file and runs NGDBuild to create a fully expanded NGD file that contains logic from the top-level design and each of the Physically Implemented Modules (PIMs). XFLOW then implements this NGD file, running MAP and PAR to create a fully expanded NCD file.

The working directory for this flow type should be the top-level design directory. You can either run the –assemble flow type from the top-level directory or use the –wd option to specify this directory. Specify the path to the PIMs directory after the –pd option. The input design file should be the NGO file for the top-level design. See the “Modular Design” chapter for more information.

Note If you do not use the –pd option, XFLOW searches the working directory for the PIM files.

Xilinx provides the following option files for use with this flow type. These files allow you to optimize your design based on different parameters.

Table 20-4 Option Files for –assemble Flow Type

Option Files	Description
fast_runtime.opt	Optimized for fastest runtimes at the expense of design performance Recommended for medium to slow speed designs
balanced.opt	Optimized for a balance between speed and high effort
high_effort.opt	Optimized for high effort at the expense of longer runtimes Recommended for creating designs that operate at high speeds

The following example shows how to assemble a Modular Design with a top-level design named “top”:

```
xflow –p xcv100bg256-5 –assemble balanced.opt
–pd ../pims top.ngo
```

–config (Create a BIT File for FPGAs)

–config *option_file*

This flow type creates a bitstream for FPGA device configuration using a routed design. It invokes the fpga.flw flow file and runs BitGen.

Xilinx provides the bitgen.opt option file for use with this flow type.

To use a netlist file as input, you must use the –implement flow type with the –config flow type. The following example shows how to use multiple flow types to implement and configure an FPGA:

```
xflow –p xcv100bg256-5 –implement balanced.opt
–config bitgen.opt testclk.edf
```

To use this flow type without the –implement or –config flow type, you must use a placed and routed NCD file as input.

–fit (Fit a CPLD)

–fit *option_file*

This flow type incorporates logic from your design into physical macrocell locations in a CPLD. It invokes the cpld.flw flow file and runs NGDBuild and the CPLD Fitter to create a JED file.

Xilinx provides the following option files for use with this flow type. These files allow you to optimize your design based on different parameters.

Note Do not use option files from previous releases of the software with the current release. Previous versions of the option files invoke the obsolete HiTop program, while the current version invokes CPLDFit.

Table 20-5 Option Files for –fit Flow Type

Option Files	Description
balanced.opt	Optimized for a balance between speed and density
speed.opt	Optimized for speed
density.opt	Optimized for density

The following example shows how to use a combination of flow types to fit a design and generate a VHDL timing simulation netlist for a CPLD:

```
xflow -p xc95144pq160-7 -fit balanced.opt -tsim
generic_vhdl.opt main_pcb.edn
```

–fsim (Create a File for Functional Simulation)

–fsim *option_file*

Note The –fsim flow type can be used alone or with the –synth flow type. It cannot be combined with the –implement, –tsim, –fit, or –config flow types.

This flow type generates a file that can be used for functional simulation of an FPGA or CPLD design. It invokes the fsim.flw flow file and runs NGDBuild and one of the netlist writers (NGD2EDIF, NGD2VER, or NGD2VHDL) to create a func_sim.edn, func_sim.v, or func_sim.vhdl file. This file contains a netlist description of your design in terms of Xilinx simulation primitives. You can use the functional simulation file to perform a back-end simulation with a simulator.

Xilinx provides the following option files, which are targeted to specific vendors, for use with this flow type.

Table 20-6 Option Files for –fsim Flow Type

Option File	Description
generic_vhdl.opt	Generic VHDL
active_vhdl.opt	Active VHDL
modelsim_vhdl.opt	Modelsim VHDL
vss_vhdl.opt	VSS VHDL
speedwave_vhdl.opt	Speedwave VHDL
generic_verilog.opt	Generic Verilog
modelsim_verilog.opt	Modelsim Verilog
concept_nc_verilog.opt	Concept-NC Verilog
concept_verilog_xl.opt	Concept Verilog-XL
nc_verilog.opt	NC Verilog
verilog_xl.opt	Verilog-XL
vcs_verilog.opt	VCS Verilog
generic_edif.opt	Generic EDIF
fndtn_edif.opt	Foundation EDIF
viewsim_edif.opt	Viewsim EDIF
quicksim_edif.opt	Quicksim EDIF

The following example shows how to generate an EDIF functional simulation netlist for an FPGA:

```
xflow -p xcv100bg256-5 -fsim generic_edif.opt
testclk.edf
```

–implement (Implement an FPGA)

```
–implement option_file
```

This flow type implements your design. It invokes the fpga.flw flow file and runs NGDBuild, MAP, PAR, and then TRACE. It outputs a placed and routed NCD file.

Xilinx provides the following option files for use with this flow type. These files allow you to optimize your design based on different parameters.

Table 20-7 Option Files for –implement Flow Type

Option Files	Description
fast_runtime.opt	Optimized for fastest runtimes at the expense of design performance Recommended for medium to slow speed designs
balanced.opt	Optimized for a balance between speed and high effort
high_effort.opt	Optimized for high effort at the expense of longer runtimes Recommended for creating designs that operate at high speeds

The following example shows how to use the –implement flow type:

```
xflow -p xcv100bg256-5 -implement balanced.opt
testclk.edf
```

–initial (Initial Budgeting of Modular Design)

```
–initial budget.opt
```

Note This flow type is supported for Virtex/-II/-II PRO/-E and Spartan-II/-IIE device families only.

This flow type runs the first phase of the Modular Design flow. In this “Initial Budgeting” phase, the team leader generates an NGO and NGD file for the top-level design. The team leader then sets up initial budgeting for the design. This includes assigning top-level timing constraints as well as location constraints for various resources, including each module.

This flow type invokes the fpga.flw flow file and runs NGDBuild to create an NGO and NGD file for the top-level design with all of the instantiated modules represented as unexpanded blocks. After running this flow type, assign constraints for your design using the the Floorplanner and Constraints Editor tools.

Note You cannot use the NGD file for mapping.

The working directory for this flow type should be the top-level design directory. You can either run the –initial flow type from the top-level design directory or use the –wd option to specify this directory. The input design file should be an EDIF netlist or an NGC netlist from XST. If you use an NGC file as your top-level design, be sure to specify the .ngc extension as part of your design name. See the "Modular Design" chapter for more information.

Xilinx provides the budget.opt option file for use with this flow type.

The following example shows how to run initial budgeting for a Modular Design with a top-level design named "top":

```
xflow –p xcv100bg256-5 –initial budget.opt
top.edf
```

–module (Active Module Implementation)

–module *option_file* **–active** *module_name*

Note This flow type is supported for Virtex/-II/-II PRO/-E and Spartan-II/-IIE device families only. You *cannot* use NCD files from previous software releases with Modular Design in the current release. You must generate new NCD files with the current release of the software.

This flow type runs the second phase of the Modular Design flow. In this "Active Module Implementation" phase, each team member creates an NGD file for his or her module, implements the NGD file to create a Physically Implemented Module (PIM), and publishes the PIM using the PIMCreate command line tool.

This flow type invokes the fpga.flw flow file and runs NGDBuild to create an NGD file with just the specified "active" module expanded. This output NGD file is named after the top-level design. XFLOW then runs MAP and PAR to create a PIM.

Then, you must run PIMCreate to publish the PIM to the PIMs directory. PIMCreate copies the local, implemented module file, including the NGO, NGM and NCD files, to the appropriate module directory inside the PIMs directory and renames the files to *module_name.extension*. To run PIMCreate, type the following on the command line or add it to your flow file:

pimcreate *pim_directory* **-ncd** *design_name*_**routed.ncd**

The working directory for this flow type should be the active module directory. You can either run the –module flow type from the active module directory or use the –wd option to specify this directory. This directory should include the active module netlist file and the top-level UCF file generated during the Initial Budgeting phase. You must specify the name of the active module after the –active option, and use the top-level NGO file as the input design file. See the "Modular Design" chapter for more information.

Xilinx provides the following option files for use with this flow type. These files allow you to optimize your design based on different parameters.

Table 20-8 Option Files for –module Flow Type

Option Files	Description
fast_runtime.opt	Optimized for fastest runtimes at the expense of design performance Recommended for medium to slow speed designs
balanced.opt	Optimized for a balance between speed and high effort
high_effort.opt	Optimized for high effort at the expense of longer runtimes Recommended for designs that operate at high speeds

The following example shows how to implement a module:

```
xflow –p xcv100bg256-5 –module balanced.opt –active
controller ~teamleader/mod_des/implemented/top/
top.ngo
```

–mppr (Multi-Pass Place and Route for FPGAs)

```
–mppr option_file
```

Note This flow type is supported for FPGA devices only.

This flow type runs multiple place and route passes on your design. It invokes the fpga.flw flow file and runs NGDBuild, MAP, multiple PAR passes, and TRACE. After running the multiple PAR passes, XFLOW saves the "best" NCD file in the subdirectory called mppr.dir. (Do not change the name of this default directory.) This NCD file uses the naming convention *placer_level_router_level_cost_table*.ncd.

XFLOW then copies this "best" result to the working directory and renames it *design_name*.ncd. It also copies the relevant DLY, PAD, PAR, and XPI files to the working directory.

Note By default, XFLOW does not support the multiple-node feature of the PAR Turns Engine. If you want to take advantage of this UNIX-specific feature, you can modify the appropriate option file to include the PAR –m option. See the "–m (Multi-Tasking Mode)" section of the "PAR" chapter for more information.

Xilinx provides the following option files for use with this flow type. These files allow you to set how exhaustively PAR attempts to place and route your design.

Note Each place and route iteration uses a different "cost table" to create a different NCD file. There are 100 cost tables numbered 1 through 100. Each cost table assigns weighted values to relevant factors such as constraints, length of connection, and available routing resources.

Table 20-9 Option Files for –mppr Flow Type

Option Files	Description
overnight.opt	Runs 10 place and route iterations
weekend.opt	Runs place and route iterations until the design is fully routed or until 100 iterations are complete
exhaustive.opt	Runs 100 place and route iterations

The following example shows how to use the -mppr flow type:

```
xflow -p xcv100bg256-5 –mppr overnight.opt
testclk.edf
```

–synth (XST Synthesis)

–synth *option_file*

Note When using the –synth flow type, you must specify the –p option.

This flow type allows you to synthesize your design for implementation in an FPGA, fitting in a CPLD, or compiling for functional simulation. The input design file can be a Verilog or VHDL file. If you have multiple VHDL or Verilog files, you can use a PRJ or V file that references these files as input.

You can use the –synth flow type alone or combine it with the –implement, –fit, or –fsim flow type. If you use the –synth flow type alone, XFLOW invokes either the fpga.flw or cpld.flw file and runs XST to synthesize your design. If you combine the –synth flow type with the –implement, –fit, or –fsim flow type, XFLOW invokes the appropriate flow file, runs XST to synthesize your design, and processes your design as described in one of the following sections:

- "–implement (Implement an FPGA)"
- "–fit (Fit a CPLD)"
- "–fsim (Create a File for Functional Simulation)"

Xilinx provides the following option files for use with the –synth flow type. These files allow you to optimize your design based on different parameters.

Table 20-10 Option Files for –synth Flow Type

Option File	Description
vhdl_area.opt	Optimizes a VHDL source file for area, which reduces the total amount of logic to improve design implementation or fitting
vhdl_speed.opt	Optimizes a VHDL source file for speed, which reduces the number of logic levels and increases the speed of the design
verilog_area.opt	Optimizes a Verilog source file for area, which reduces the total amount of logic to improve design implementation or fitting
verilog_speed.opt	Optimizes a Verilog source file for speed, which reduces the number of logic levels and increases the speed of the design

The following example shows how to use to use a combination of flow types to synthesize and implement a design:

```
xflow –p xcv100bg256-5 –synth vhdl_area.opt
-implement balanced.opt testclk.prj
```

–tsim (Create a File for Timing Simulation)

```
–tsim option_file
```

This flow type generates a file that can be used for timing simulation of an FPGA or CPLD design. It invokes the fpga.flw or cpld.flw flow file, depending on your target device. For FPGAs, it runs NGDAnno and one of the netlist writers (NGD2EDIF, NGD2VER, or NGD2VHDL). For CPLDs, it runs TSim and one of the netlist writers (NGD2EDIF, NGD2VER, or NGD2VHDL). This creates a time_sim.edn, time_sim.v, or time_sim.vhdl file that contains a netlist description of your design in terms of Xilinx simulation primitives.

You can use the timing simulation file to perform a back-end simulation with a simulator.

Xilinx provides the following option files, which are targeted to specific vendors, for use with this flow type.

Table 20-11 Option Files for –tsim Flow Type

Option File	Description
generic_vhdl.opt	Generic VHDL
active_vhdl.opt	Active VHDL
modelsim_vhdl.opt	Modelsim VHDL
vss_vhdl.opt	VSS VHDL
speedwave_vhdl.opt	Speedwave VHDL
generic_verilog.opt	Generic Verilog
modelsim_verilog.opt	Modelsim Verilog
concept_nc_verilog.opt	Concept-NC Verilog
concept_verilog_xl.opt	Concept Verilog-XL
nc_verilog.opt	NC Verilog
verilog_xl.opt	Verilog-XL
vcs_verilog.opt	VCS Verilog
generic_edif.opt	Generic EDIF
fndtn_edif.opt	Foundation EDIF
viewsim_edif.opt	Viewsim EDIF
quicksim_edif.opt	Quicksim EDIF

The following example shows how to use a combination of flow types to fit and perform a VHDL timing simulation on a CPLD:

```
xflow -p xc95144pq160-7 -fit balanced.opt -tsim
generic_vhdl.opt main_pcb.edn
```

Flow Files

When you specify a flow type on the command line, XFLOW invokes the appropriate flow file and executes some or all of the programs listed in the flow file. Programs are run in the order specified in the flow file. These files have a .flw extension.

Xilinx provides three flow files. You can edit these flow files, to add a new program, modify the default settings, or add your own commands between Xilinx programs. However, you cannot create new flow files of your own.

The following table lists the flow files invoked for each flow type.

Table 20-12 Xilinx Flow Files

Flow Type	Flow File	Devices	Flow Phase	Programs Run
–synth	fpga.flw	FPGA	Synthesis	XST
–initial			Modular Design Initial Budgeting Phase	NGDBuild
–module			Modular Design Active Module Implementation Phase	NGDBuild, MAP, PAR
–assemble			Modular Design Final Assembly Phase	NGDBuild, MAP, PAR
–implement			Implementation	NGDBuild, MAP, PAR, TRACE
–mppr			Implementation (with Multi-Pass Place and Route)	NGDBuild, MAP, PAR (multiple passes), TRACE
–tsim			Timing Simulation	NGDAnno One of the following: NGD2EDIF, NGD2VER, NGD2VHDL
–config			Configuration	BitGen
–synth	cpld.flw	CPLD	Synthesis	XST
–fit			Fit	NGDBuild, CPLDFit, TAEngine, HPREP6
–tsim			Timing Simulation	TSim One of the following: NGD2EDIF, NGD2VER, NGD2VHDL
–synth	fsim.flw	FPGA / CPLD	Synthesis	XST
–fsim			Functional Simulation	NGDBuild One of the following: NGD2EDIF, NGD2VER, NGD2VHDL

Flow File Format

The flow file is an ASCII file that contains the following information:

Note You can use variables for the file names listed on the Input, Triggers, Export, and Report lines. For example, if you specify **Input: <design>.edf** on the Input line, XFLOW automatically reads the EDIF file in your working directory as the input file.

- ExportDir

  This section specifies the directory in which to copy the output files of the programs in the flow. The default directory is your working directory.

  Note You can also specify the export directory using the –ed command line option. The command line option overrides the ExportDir specified in the flow file.

- ReportDir

  This section specifies the directory in which to copy the report files generated by the programs in the flow. The default directory is your working directory.

  Note You can also specify the report directory using the –rd command line option. The command line option overrides the ReportDir specified in the flow file.

- Global user-defined variables

  This section allows you to specify a value for a global variable, as shown in the following example:

  ```
  Variables
  $simulation_output = time_sim;
  End variables
  ```

The flow file contains a program block for each program in the flow. Each program block includes the following information:

- **Program** *program_name*

  This line identifies the name of the program block. It also identifies the command line executable if you use an executable name as the *program_name*, for example, ngdbuild. This is the first line of the program block.

- **Flag: ENABLED | DISABLED**
  - ENABLED: This option instructs XFLOW to run the program if there are options in the options file.
  - DISABLED: This option instructs XFLOW to *not* run the program even if there are corresponding options in the options file.
- **Input:** *filename*

  This line lists the name of the input file for the program. For example, the NGDBuild program block might list design.edn.
- **Triggers:**

  This line lists any additional files that should be read by the program. For example, the NGDBuild program block might list design.ucf.
- **Exports:**

  This line lists the name of the file to export. For example, the NGDBuild program block might list design.ngd.
- **Reports:**

  This line lists the report files generated. For example, the NGDBuild program block might list design.bld.
- **Executable:** *executable_name*

  This line is optional. It allows you to create multiple program blocks for the same program. When creating multiple program blocks for the same program, you must enter a name other than the program name in the Program line (for example, enter post_map_trace, not trce). In the Executable line, you enter the name of the program as you would enter it on the command line (for example, trce).

For example, if you want to run TRACE after MAP and again after PAR, the program blocks for post-MAP TRACE and post-PAR TRACE appear as follows:

```
Program post_map_trce
Flag: ENABLED;
Executable: trce;
Input: <design>_map.ncd;
Exports: <design>.twr, <design>.tsi;
End Program post_map_trce


Program post_par_trce
Flag: ENABLED;
Executable: trce;
Input: <design>.ncd;
Reports: <design>.twr, <design>.tsi;
End Program post_par_trce
```

Note If your option file includes a corresponding program block, its Program line must match the Program line in the flow file (for example, post_map_trace).

- **End Program** *program_name*

  This line identifies the end of a program block. The *program_name* should be consistent with the *program_name* specified on the line that started the program block.

User Command Blocks

To run your own programs in the flow, you can add a “user command block” to the Flow File. The syntax for a user command block is the following:

```
UserCommand
   Cmdline: <user_cmdline>;
End UserCommand
```

Following is an example:

```
UserCommand

   Cmdline: “myscript.csh”;
End UserCommand
```

Note You cannot use the asterisk “*,” dollar sign “$,” or parentheses “( )” character as part of your command line command.

XFLOW Option Files

Option files contain the options for all programs run in a flow. These files have an .opt extension. Xilinx provides option files for each flow type, as described in the different sections of the “XFLOW Flow Types” section. You can also create your own option files.

Note If you want to create your own option files, Xilinx recommends that you make a copy of an existing file, rename it, and then modify it.

Option File Format

Option files are in ASCII format. They contain program blocks that correspond to the programs listed in the flow files. Option file program blocks list the options to run for each program. Program options can be command line options or parameter files.

- Command Line Options

  For information on the different command line options for each command line program, see the various chapters of this guide, or type the program name followed by –h on the command line. Some options require that you specify a particular file or value.

- Parameter files

  Parameter files specify parameters for a program. Parameters are written into the specified file. For example, Xilinx Synthesis Technology (XST) uses a script file to execute its command line options:

```
Program xst
     -ifn <design>_xst.scr;
     -ofn <design>_xst.log;
     ParamFile: <design>_xst.scr
         "run";
         "-ifn <synthdesign>";
         "-ifmt Verilog";
         "-ofn <design>.edn";
.
.
.
     End ParamFile
End Program xst
```

Note You can use variables for the file names listed in the Option Files. For example, if you specify **<design>.edf** as an input file, XFLOW automatically reads the EDF file in your working directory as the input file.

XFLOW Options

This section describes the XFLOW command line options. These options can be used with any of the flow types described in the preceding section.

–ed (Copy Files to Export Directory)

```
–ed export_directory
```

The –ed option copies files listed in the Export line of the flow file to the directory you specify. If you do not use the –ed option, the files are copied to the working directory. See the "Flow Files" section for a description of the Export line of the flow file.

If you use the –ed option with the –wd option and do not specify an absolute path name for the export directory, the export directory is placed underneath the working directory.

In the following example, the export3 directory is created underneath the sub3 directory:

```
xflow -implement balanced.opt -wd sub3 -ed export3
testclk.edf
```

If you do not want the export directory to be a subdirectory of the working directory, enter an absolute path name as in the following example:

```
xflow -implement balanced.opt -wd sub3 -ed /usr/
export3 testclk.edf
```

–f (Execute Commands File)

```
–f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–g (Specify a Global Variable)

```
–g variable:value
```

The –g option allows you to assign a value to a variable in a flow or option file. This value is applied globally. The following example shows how to specify a global variable at the command line:

```
xflow -implement balanced -g
$simulation_output:time_sim calc
```

Note If a global variable is specified both on the command line and in a flow file, the command line takes precedence over the flow file.

–log (Specify Log File)

The –log option allows you to specify a log filename at the command line. XFLOW writes the log file to the working directory after each run. By default, the log filename is xflow.log.

–norun (Creates a Script File Only)

By default, XFLOW runs the programs enabled in the flow file. Use the –norun option if you do not want run the programs but instead want to create a script file (SCR or BAT extension). XFLOW copies the appropriate flow and option files to your working directory and creates a script file based on these files. This is useful if you want to check the programs and options listed in the script file before executing them.

Following is an example:

```
xflow -implement balanced.opt -norun testclk.edf
```

In this example, XFLOW copies the balanced.opt and fpga.flw files to the current directory and creates the following script file:

```
###########################################
# Script file to run the flow
#
###########################################
#
# Command line for ngdbuild
#
ngdbuild -p xcv100bg256-5 -nt timestamp /home/
xflow_test/testclk.edf testclk.ngd
#
# Command line for map
#
map -o testclk_map.ncd testclk.ngd testclk.pcf
#
# Command line for par
#
par -w -ol 2 -d 0 testclk_map.ncd testclk.ncd
testclk.pcf
#
# Command line for post_par_trce
#
trce -e 3 -o testclk.twr testclk.ncd testclk.pcf
```

–o (Change Output File Name)

–o *output_filename*

This option allows you to change the output file base name. If you do not specify this option, the output file name has the base name as the input file in most cases.

The following example show how to use the –o option to change the base name of output files from “testclk”to “newname”:

```
xflow -implement balanced.opt -o newname
testclk.edf
```

–p (Part Number)

```
-p part
```

By default (without the –p option), XFLOW searches for the part name in the input design file. If XFLOW finds a part number, it uses that number as the target device for the design. If XFLOW does not find a part number in the design input file, it prints an error message indicating that a part number is missing.

The –p option allows you to specify a device. For a list of valid ways to specify a part, see the "–p (Part Number)" section of the "Introduction" chapter.

For FPGA part types, you must designate a part name with a package name. If you do not, XFLOW halts at MAP and reports that a package needs to be specified. You can use the PARTGen –i option to obtain package names for installed devices. See the "–i (Print a List of Devices, Packages, and Speeds)" section of the "PARTGen" chapter for information.

For CPLD part types, either the part number or the family name can be specified.

The following example show how to use the –p option for a Virtex design:

```
xflow -p xcv100bg256-5 -implement high_effort.opt
testclk.edf
```

Note If you are running the Modular Design flow and are targeting a part different from the one specified in your source design, you must specify the part type using the –p option *every time* you run the –initial, –module, or –assemble flow type.

–rd (Copy Report Files)

```
-rd report_directory
```

The –rd option copies the report files output during the XFLOW run from the working directory to the specified directory. The original report files are kept intact in the working directory.

You can create the report directory prior to using this option, or specify the name of the report directory and let XFLOW create it for you. If you do not specify an absolute path name for the report directory, XFLOW creates the specified report directory in your working directory. Following is an example in which the report directory (reportdir) is created in the working directory (workdir):

```
xflow -implement balanced.opt -wd workdir -rd
reportdir testclk.edf
```

If you do not want the report directory to be a subdirectory of the working directory, enter an absolute path name, as shown in the following example:

```
xflow -implement balanced.opt -wd workdir -rd /usr/
reportdir testclk.edf
```

–wd (Specify a Working Directory)

–wd *working_directory*

The default behavior of XFLOW (without the –wd option) is to use the directory from which you invoked XFLOW as the working directory. The –wd option allows you to specify a different directory as the working directory. XFLOW searches for all flow files, option files, and input files in the working directory. It also runs all subprograms and outputs files in this directory.

Note If you use the –wd option and want to use a UCF file as one of your input files, you must copy the UCF file into the working directory.

Unless you specify a directory path, the working directory is created in the current directory. For example, if you enter the following command, the directory sub1 is created in the current directory:

```
xflow -fsim generic_edif.opt -wd sub1 testclk.edf
```

You can also enter an absolute path for a working directory as in the following example. You can specify an existing directory or specify a path for XFLOW to create.

```
xflow -fsim generic_edif.opt -wd /usr/project1
testclk.edf
```

Running XFLOW

The following sections describe common ways to use XFLOW.

Using XFLOW Flow Types in Combination

You can combine flow types on the XFLOW command line to run different flows.

The following example shows how to use a combination of flow types to implement a design, create a bitstream for FPGA device configuration, and generate an EDIF timing simulation netlist for an FPGA design named testclk:

```
xflow -p xcv100bg256-5 -implement balanced -tsim
generic_edif -config bitgen testclk
```

The following example shows how to use a combination of flow types to fit a CPLD design and generate a VHDL timing simulation netlist for a CPLD design named main_pcb:

```
xflow -p xc95144pq160-7 -fit balanced -tsim
generic_vhdl main_pcb
```

Running “Smart XFLOW”

“Smart XFLOW” automatically detects changes to your input files and runs the flow from the appropriate point. XFLOW detects changes made to design files, flow files, option files, and trigger files. It also detects and reruns aborted flows. To run “Smart XFLOW,” type the XFLOW syntax *without* specifying an extension for your input design. XFLOW automatically detects which input file to read and starts the flow at the appropriate point.

For example, if you enter the following command and XFLOW detects changes to the calc.edf file, XFLOW runs *all* the programs in the flow and option files. However, if you enter the same command and XFLOW detects changes only to the calc.mfp file generated by the Floorplanner GUI, XFLOW starts the flow with the MAP program.

```
xflow -implement balanced.opt calc
```

Using the SCR or BAT File

Every time you run XFLOW, it creates a script file that includes the command line commands of all the programs run. You can use this file for the following:

- Review this file to check which commands were run.
- Execute this file instead of running XFLOW.

By default, this file is named xflow.bat (PC) or xflow.scr (UNIX). To execute the script file, type **`xflow.bat`** or **`xflow.scr`** at the command line.

If you choose to execute the script file instead of using XFLOW, the features of "Smart XFLOW" are not enabled. For example, XFLOW starts the flow at an appropriate point based on which files have changed, while the script file simply runs every command listed in the file. In addition, the script file does not provide error detection. For example, if an error is encountered during NGDBuild, XFLOW detects the error and terminates the flow, while the script file continues and runs MAP.

Using the XIL_XFLOW_PATH Environment Variable

This environment variable is useful for team-based design. By default, XFLOW looks for all flow and option files in your working directory. However, this variable allows you to store flow and option files in a central location and copy them to your team members' local directories, ensuring consistency. To use this variable, do the following:

1. Modify the flow and option files as necessary.
2. Copy the flow and option files to the central directory, and provide your team members with the directory location.
3. Instruct your team members to type the following from their working directory:

   ```
   set XIL_XFLOW_PATH=name_of_central_directory
   ```

   When the team member runs XFLOW, XFLOW copies all flow and option files from the central directory to his or her local directory.

Note If you alter the files in the central directory and want to repopulate the users' local directories, they must delete their local copies of the flow and option files, set the XIL_FLOW_PATH environment variable, and rerun XFLOW to copy in the updated files.

Halting XFLOW

You can manually interrupt the flow while XFLOW is in session by typing **`Ctrl C`** on your keyboard. If you halt XFLOW while PAR is in progress, you can choose one of several options. See the "Halting PAR" section of the "PAR" chapter for a detailed description.

Appendix A

Xilinx Development System Files

This appendix gives an alphabetic listing of the files used by the Xilinx Development System.

Name	Type	Produced By	Description
ALF	ASCII	NGDAnno	Log file containing information about an NGDAnno run
ARF	ASCII	NGDAnno	Report file containing information about lost instance or net names
BIT	Data	BitGen	Download bitstream file for devices containing all of the configuration information from the NCD file
BGN	ASCII	BitGen	Report file containing information about a BitGen run
BLD	ASCII	NGDBuild	Report file containing information about an NGDBuild run, including the subprocesses run by NGDBuild
DATA	C File	TRCE	File created with the –stamp option to TRCE that contains timing model information
DC	ASCII	Synopsys FPGA Compiler	Synopsys setup file containing constraints read into the Xilinx Development System
DLY	ASCII	PAR	File containing delay information for each net in a design
DRC	ASCII	BitGen	Design Rule Check file produced by BitGen

Name	Type	Produced By	Description
EDIF (various file extensions)	ASCII	CAE vendor's EDIF 2 0 0 netlist writer.	EDIF netlist. The Xilinx Development System accepts an EDIF 2 0 0 Level 0 netlist file
EDN	ASCII	NGD2EDIF	Default extension for an EDIF 2 0 0 netlist file
EPL	ASCII	FPGA Editor	FPGA Editor command log file. The EPL file keeps a record of all FPGA Editor commands executed and output generated. It is used to recover an aborted FPGA Editor session.
EXO	Data	PROMGen	PROM file in Motorola's EXORMAT format
FLW	ASCII	Provided with software	File containing command sequences for XFLOW programs
fpga_editor.ini	ASCII	Xilinx software	Script that determines what FPGA Editor commands are performed when the FPGA Editor starts up
fpga_editor_user.ini	ASCII	Xilinx software	Supplement to the fpga_editor.ini file used for modifying or adding to the fpga_editor.ini file
GYD	ASCII	CPLD fitter	CPLD guide file
HEX	Hex	PROMGen Command	Output file from PROMGEN that contains a hexadecimal representation of a bitstream
IBS	ASCII	IBISWriter Command	Output file from IBISWriter that consists of a list of pins used by the design, the signals internal to the device that connect to those pins, and the IBIS buffer models for the IOBs connected to the pins
ITR	ASCII	PAR	Intermediate failing timespec summary from routing
JED	JEDEC	CPLD fitter	Programming file to be downloaded to a device

Name	**Type**	**Produced By**	**Description**
LOG	ASCII	NGD2VER NGD2VHDL XFLOW	Log file containing all the messages generated during the execution of NGD2VER (ngd2ver.log), NGD2VHDL (ngd2vhdl.log), or XFLOW (xflow.log)
LL	ASCII	BitGen	Optional ASCII logic allocation file with an .ll extension. The logic allocation file indicates the bitstream position of latches, flip-flops, and IOB inputs and outputs.
MEM	ASCII	User (with text editor) LogiBLOX	User-edited memory file that defines the contents of a ROM
MCS	Data	PROMGen	PROM-formatted file in Intel's MCS-86 format
MDF	ASCII	MAP	A file describing how logic was decomposed when the design was mapped. The MDF file is used for guided mapping.
MFP	ASCII	Floorplanner	Map Floorplanner File, which is generated by the Floorplanner, specified as an input file with the –fp option. The MFP file is essentially used as a guide file for mapping.
MOD	ASCII	TRCE	File created with the –stamp option in TRCE that contains timing model information
MRP	ASCII	MAP	MAP report file containing information about a technology mapper command run
MSK	Data	BitGen	File used to compare relevant bit locations when reading back configuration data contained in an operating Xilinx device

Name	Type	Produced By	Description
NAV	XML	NGDBuild	Report file containing information about an NGDBuild run, including the subprocesses run by NGDBuild. From this file, the user can click any linked net or instance names to navigate back to the net or instance in the source design.
NCD	Data	Mappers, PAR, FPGA Editor	Flat physical design database correlated to the physical side of the NGD in order to provide coupling back to the user’s original design
NCF	ASCII	CAE Vendor toolset	Vendor-specified logical constraints files
NGA	Data	NGDAnno	Back-annotated mapped NCD file
NGC	Binary	LogiBLOX	File containing the implementation of a module in the design
		XST	Netlist file with constraint information
NGD	Data	NGDBuild	Generic Database file. This file contains a logical description of the design expressed both in terms of the hierarchy used when the design was first created and in terms of lower-level Xilinx primitives to which the hierarchy resolves.
NGM	Data	MAP	File containing all of the data in the input NGD file as well as information on the physical design produced by the mapping. The NGM file is used for back-annotation.
NGO	Data	Netlist Readers	File containing a logical description of the design in terms of its original components and hierarchy
NKY	Data	BitGen	Encryption key file

Name	Type	Produced By	Description
NMC	Binary	FPGA Editor	Xilinx physical macro library file containing a physical macro definition that can be instantiated into a design
OPT	Text	Input file option	Option file used by XFLOW
PAD	ASCII	PAR	File containing a listing of all I/O components used in the design and their associated primary pins
PAR	ASCII	PAR	PAR report file containing execution information about the PAR command run. The file shows the steps taken as the program converges on a placement and routing solution
partlist.xct	ASCII	PARTGen	File containing detailed information about architectures and devices
PCF	ASCII	MAP, FPGA Editor	File containing physical constraints specified during design entry (that is, schematics) and constraints added by the user
PIN	ASCII	NGD2VER	Cadence signal-to-pin mapping file
PRM	ASCII	PROMGen	File containing a memory map of a PROM file showing the starting and ending PROM address for each BIT file loaded
RBT	ASCII	BitGen	“Rawbits" file consisting of ASCII ones and zeros representing the data in the bitstream file
RPT	ASCII	PIN2UCF	Report file generated by PIN2UCF when conflicting constraints are discovered. The name is pinlock.rpt.
RCV	ASCII	FPGA Editor	FPGA Editor recovery file
SCR	ASCII	FPGA Editor or XFLOW	FPGA Editor or XFLOW command script file
SDF	ASCII	NGD2VER, NGD2VHDL	File containing the timing data for a design. Standard Delay Format File

Name	Type	Produced By	Description
TDR	ASCII	DRC	Physical DRC report file
TEK	Data	PROMGen	PROM-formatted file in Tektronix's TEKHEX format
TV	ASCII	NGD2VER	Verilog test fixture file
TVHD	ASCII	NGD2VHDL	VHDL testbench file
TWR	ASCII	TRACE	Timing report file produced by TRACE
TWX	XML	TRACE	Timing report file produced by TRACE. From this file, the user can click any linked net or instance names to navigate back to the net or instance in the source design.
UCF	ASCII	User (with text editor)	User-specified logical constraints files
URF	ASCII	User (with text editor)	User-specified rules file containing information about the acceptable netlist input files, netlist readers, and netlist reader options
V	ASCII	NGD2VER	Verilog netlist
VHD	ASCII	NGD2VHDL	VHDL netlist
VM6	Design	CPLD Fitter	Output file from fitter
XMM	ASCII	NGD2EDIF	File defining the initial contents of the RAMs in the design for a simulator
XNF	ASCII	Previous releases of Xilinx Development System, CAE vendor toolsets	Xilinx netlist format file
XTF	ASCII	Previous releases of Xilinx Development System	Xilinx netlist format file
XPI	ASCII	PAR	File containing PAR run summary

Appendix B

EDIF2NGD, XNF2NGD, and NGDBuild

This appendix describes the netlist reader programs, EDIF2NGD and XNF2NGD, and how these programs interact with NGDBuild. The appendix contains the following sections:

- "EDIF2NGD"
- "XNF2NGD"
- "NGDBuild"
- "Netlister Launcher"
- "NGDBuild File Names and Locations"

EDIF2NGD

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

The EDIF2NGD program allows you to read an EDIF (Electronic Design Interchange Format) 2 0 0 file into the Xilinx Development System toolset. EDIF2NGD converts an industry-standard EDIF netlist to an NGO file—a Xilinx-specific format. The EDIF file includes the hierarchy of the input schematic. The output NGO file is a binary database describing the design in terms of the components and hierarchy specified in the input design file. After you convert the EDIF file to an NGO file, you run NGDBuild to create an NGD file, which expands the design to include a description reduced to Xilinx primitives.

The following figure shows the flow through EDIF2NGD.

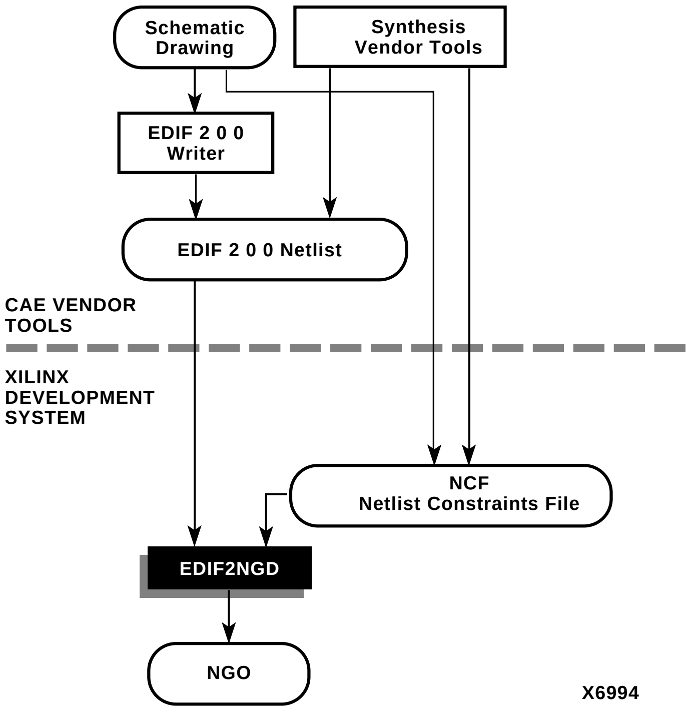


Figure B-1 EDIF2NGD Design Flow

You can run EDIF2NGD in the following ways:

- Automatically from NGDBuild
- From the UNIX or DOS command line, as described in the following sections

Note When creating nets or symbols names, do not use reserved names. Reserved names are the names of symbols for primitives and macros in the *Libraries Guide* and net names GSR, RESET, GR, and PRELOAD. If you used these names, EDIF2NGD issues an error.

EDIF2NGD Syntax

The following command reads your EDIF netlist and converts it to an NGO file:

```
edif2ngd [options] edif_file ngo_file
```

options can be any number of the EDIF2NGD options listed in the "EDIF2NGD Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

edif_file is the EDIF 2 0 0 input file to be converted. If you enter a file name with no extension, EDIF2NGD looks for a file with the name you specified and an .edn extension. If the file has an extension other than .edn, you must enter the extension as part of *edif_file*.

Note For EDIF2NGD to read a Mentor Graphics EDIF file, you must have installed the Mentor Graphics software component on your system. Similarly, to read a Cadence EDIF file, you must have installed the Cadence software component.

ngo_file is the output file in NGO format. The output file name, its extension, and its location are determined in the following ways:

- If you do not specify an output file name, the output file has the same name as the input file, with an .ngo extension.
- If you specify an output file name with no extension, EDIF2NGD appends the .ngo extension to the file name.
- If you specify a file name with an extension other than .ngo, you get an error message and EDIF2NGD does not run.
- If you do not specify a full path name, the output file is placed in the directory from which you ran EDIF2NGD.

If the output file exists, it is overwritten with the new file.

EDIF2NGD Input Files

EDIF2NGD uses the following files as input:

- EDIF file—This is an EDIF 2 0 0 netlist file. The file must be a Level 0 EDIF netlist, as defined in the EDIF 2 0 0 specification. The Xilinx Development System toolset can understand EDIF files developed using components from any of these libraries:
  - Xilinx Unified Libraries (described in the *Libraries Guide*)
  - XSI (Xilinx Synopsys Interface) Libraries
  - Any Xilinx physical macros you create

  Note Xilinx tools do not recognize Xilinx Unified Libraries components defined as macros; they only recognize the primitives from this library. The third-party EDIF writer must include definitions for all macros.
- NCF file—This Netlist Constraints File is produced by a vendor toolset and contains constraints specified within the toolset. EDIF2NGD reads the constraints in this file and adds the constraints to the output NGO file.

  EDIF2NGD reads the constraints in the NCF file if the NCF file has the same base name as the input EDIF file and an .ncf extension. The name of the NCF file does not have to be entered on the EDIF2NGD command line.

EDIF2NGD Output Files

The output of EDIF2NGD is an NGO file—a binary file containing a logical description of the design in terms of its original components and hierarchy.

EDIF2NGD Options

This section describes the EDIF2NGD command line options.

–a (Add PADs to Top-Level Port Signals)

The –a option adds PAD properties to all top-level port signals. This option is necessary if the EDIF2NGD input is an EDIF file in which PAD symbols were translated into ports. If you do not specify a –a option for one of these EDIF files, the absence of PAD instances in the

EDIF file causes EDIF2NGD to read the design incorrectly. Subsequently, MAP interprets the logic as unused and removes it.

In all Mentor Graphics and Cadence EDIF files PAD symbols are translated into ports. For EDIF files from either of these vendors, the –a option is set automatically; you do not have to enter the –a option on the EDIF2NGD command line.

–aul (Allow Unmatched LOCs)

By default (without the –aul option), EDIF2NGD generates an error if the constraints specified for pin, net, or instance names in the NCF file cannot be found in the design. If this error occurs, an NGO file is not written. If you enter the –aul option, EDIF2NGD generates a warning instead of an error for LOC constraints and writes an NGO file.

You may want to run EDIF2NGD with the –aul option if your constraints file includes location constraints for pin, net, or instance names that have not yet been defined in the HDL or schematic. This allows you to maintain one version of your constraints files for both partially complete and final designs.

Note When using this option, make sure you do not have misspelled net or instance names in your design. Misspelled names may cause inaccurate placing and routing.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–l (Libraries to Search)

```
-l libname
```

The –l option specifies a library to search when determining what library components were used to build the design. This information is necessary for NGDBuild, which must determine the source of the design's components before it can resolve the components to Xilinx primitives.

You may specify multiple –l options on the command line. Each must be preceded with –l; you cannot combine multiple *libname* specifiers after one –l. For example, **–l xilinxun synopsys** is not acceptable, while **–l xilinxun –l synopsys** is acceptable.

The allowable entries for *libname* are the following.

- **xilinxun** (For Xilinx Unified library)
- **synopsys**

Note You do not have to enter xilinxun with a –l option. The Xilinx Development System tools automatically access these libraries. You do not have to enter synopsys with a –l option if the EDIF netlist contains an author construct with the string "Synopsys." In this case, EDIF2NGD automatically detects that the design is from Synopsys.

–p (Part Number)

```
-p part
```

The –p option specifies the part into which your design is implemented. The –p option can specify an architecture only, a complete part specification (device, package, and speed), or a partial specification (for example, device and package only).

The syntax for the –p option is described in the "–p (Part Number)" section of the "Introduction" chapter. Examples of *part* entries are **XCV50-TQ144** and **XCV50-TQ144-5**.

If you do not specify a part when you run EDIF2NGD, you must specify one when you run NGDBuild.

You can also use the –p option to override a part name in the input EDIF netlist or a part name in an NCF file.

–quiet (Report Warnings and Errors Only)

The –quiet option reduces EDIF2NGD screen output to warnings and errors only. This option is useful if you only want a summary of the EDIF2NGD run.

–r (Ignore LOC Constraints)

The –r option filters out all location constraints (LOC=) from the design. This option can be used when you are migrating to a different device or architecture, because locations in one architecture do not match locations in another.

XNF2NGD

This program is compatible with the following families:

- Spartan / XL
- XC4000E / L / EX / XL / XLA

Note XNF primitives are not defined for the Virtex families, and XNF files created for Virtex families are rejected by XNF2NGD. However, if you have XNF netlists that were created for theXC4000E architectures, you can include these XNF netlists in a design that you target to a Virtex device.

XNF2NGD allows you to read a Version 6.1 XNF (Xilinx Netlist Format) file into the Xilinx Development System toolset. XNF2NGD converts an XNF file to an NGO file, which is a binary database describing the netlist in terms of Xilinx components. After you convert the XNF file to an NGO file, you run NGDBuild to create an NGD file, which expands the design to include a description reduced to Xilinx primitives.

The following figure shows the flow through XNF2NGD.

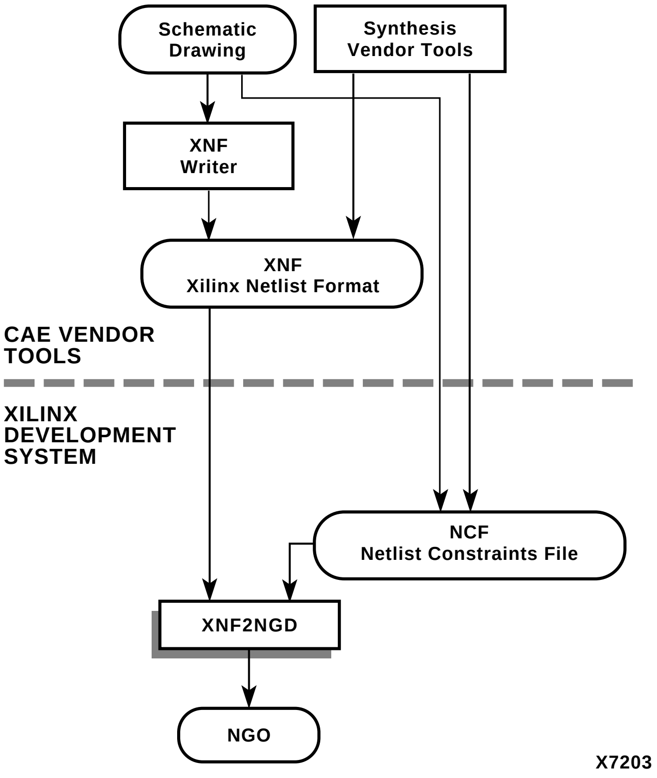


Figure B-2 XNF2NGD Design Flow

You can run XNF2NGD in the following ways:

- Automatically from NGDBuild
- From the UNIX or DOS command line, as described in the following sections

Note When creating nets or symbols names, do not use reserved names. Reserved names are the names of symbols for primitives and macros in the *Libraries Guide* and net names GSR,RESET, GR, and PRELOAD. If you use these names, XNF2NGD issues an error.

XNF2NGD Syntax

The following command reads your XNF netlist and converts it to an NGO file:

```
xnf2ngd [options] xnf_file ngo_file
```

options can be any number of the XNF2NGD options listed in the "XNF2NGD Options" section. They do not need to be listed in any particular order. Separate multiple options with spaces.

xnf_file is the input file (in XNF format) to be converted. The file can have any extensions (for example, .xnf, .xtf, .xff, .xg, or .sxnf), as long as the file is in XNF format. If you enter a file name with no extension, XNF2NGD looks for a file with an .xnf extension and the name you specified.

ngo_file is the output file in NGO format. The output file name, its extension, and its location are determined in the following ways:

- If you do not specify an output file name, the output file has the same name as the input file, with an .ngo extension.
- If you specify an output file name with no extension, XNF2NGD appends the .ngo extension to the file name.
- If you specify a file name with an extension other than .ngo, you get an error message and XNF2NGD does not run.
- If you do not specify a full path name, the output file is placed in the directory from which you ran XNF2NGD.

If the output file already exists, it is overwritten with the new file.

XNF2NGD Input Files

XNF2NGD uses the following files as input:

- XNF file—This is the Xilinx Netlist Format (XNF) text file. The file can have any extension as long as the contents are in XNF format.
- NCF file—This Netlist Constraints File is produced by a vendor toolset and contains constraints specified within the toolset. XNF2NGD reads the constraints in this file and adds the constraints to the output NGO file.

XNF2NGD reads the constraints in the NCF file if the NCF file has the same name as the input XNF file and an extension of .ncf. The name of the NCF file does not have to be entered on the XNF2NGD command line.

XNF2NGD Output Files

The output of XNF2NGD is an NGO file—a binary file containing a logical description of the design in terms of its original components and hierarchy.

XNF2NGD Options

This section describes the XNF2NGD command line options.

–f (Execute Commands File)

```
-f command_file
```

The –f option executes the command line arguments in the specified *command_file*. For more information on the –f option, see the "–f (Execute Commands File)" section of the "Introduction" chapter.

–l (Libraries to Search)

```
-l libname
```

The –l option indicates the list of libraries to search when determining what library components were used to build the design. This information is necessary for NGDBuild, which must determine the source of the design's components before it can resolve the components to Xilinx primitives.

You can specify multiple –l options on the command line. Each must be preceded with –l; you cannot combine multiple *libname* specifiers after one -l. For example, **–l xilinxun synopsys** is not acceptable, while **–l xilinxun –l synopsys** is acceptable.

The allowable entries for *libname* are the following.

- **xilinxun** (For Xilinx Unified library)
- **synopsys**
- **XC4000**

Note You do not have to enter xilinxun with a –l option. The Xilinx Development System tools automatically access these libraries. In most cases, you do not have to enterXC4000 with a –l option. However, if your XNF file contains an input latch (INLAT) component and no part type is specified in the XNF file, the meaning of the INLAT component is ambiguous. In this case, XNF2NGD stops with an error message. You must run XNF2NGD again using the –l option to define the INLAT component; –l XC4000 means it is transparent Low.

–p (Part Number)

```
–p part
```

The –p option specifies the part into which your design is implemented. The –p option can specify an architecture only, a complete part specification (device, package, and speed), or a partial specification (for example, device and package only).

The syntax for the –p option is described in the "–p (Part Number)" section of the "Introduction" chapter. Examples of *part* entries are **XCV50-TQ144** and **XCV50-TQ144-5**.

If you do not specify a part when you run XNF2NGD, you must specify one when you run NGDBuild.

You may also use the –p option to override a part name in the input XNF netlist or a part name in an NCF file.

–r (Ignore LOC Constraints)

The –r option filters out all location constraints (LOC=) from the design. This can be used when you are migrating to a different device or architecture, because locations in one architecture do not match locations in another.

–u (Top-Level XNF Netlist)

The –u option specifies that the input XNF file is the top-level design netlist. When XNF2NGD translates netlists at lower hierarchical levels, XNF2NGD adds to the lower-level NGO file information that is unnecessary in the top-level NGO file. The –u option prevents this information from being added to the top-level NGO file.

NGDBuild

This program is compatible with the following families:

- Virtex / -II / -II PRO / -E
- Spartan / -II / -IIE / XL
- XC4000E / L / EX / XL / XLA
- CoolRunner XPLA3 / -II
- XC9500 / XL / XV

NGDBuild performs all the steps necessary to read a netlist file in XNF or EDIF format and create an NGD file describing the logical design. The NGD file resulting from an NGDBuild run contains both a logical description of the design reduced to NGD primitives and a description in terms of the original hierarchy expressed in the input netlist. The output NGD file can be mapped to the desired device family.

Converting a Netlist to an NGD File

The following figure shows the NGDBuild conversion process.

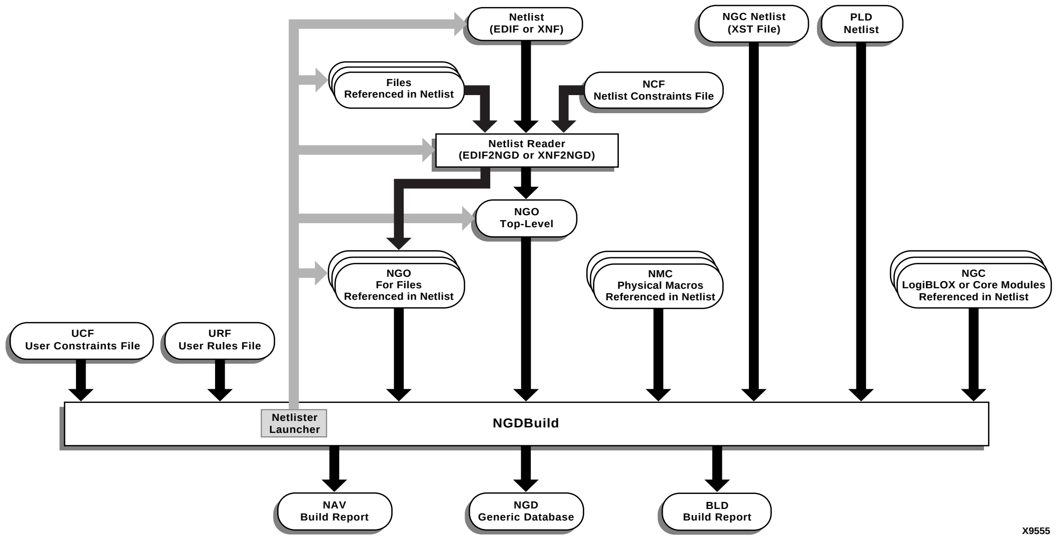

Netlist (EDIF or XNF)
NGC Netlist (XST File)
PLD Netlist
Files Referenced in Netlist
NCF Netlist Constraints File
Netlist Reader (EDIF2NGD or XNF2NGD)
NGO Top-Level
NGO For Files Referenced in Netlist
NMC Physical Macros Referenced in Netlist
NGC LogiBLOX or Core Modules Referenced in Netlist
UCF User Constraints File
URF User Rules File
Netlister Launcher
NGDBuild
NAV Build Report
NGD Generic Database
BLD Build Report
X9555

Figure B-3 NGDBuild and the Netlist Readers

NGDBuild performs the following steps to convert a netlist to an NGD file:

1. Reads the source netlist

   To perform this step, NGDBuild invokes the Netlister Launcher, a part of the NGDBuild software which determines the type of the input netlist and starts the appropriate netlist reader program. If the input netlist is in EDIF or XNF format, the Netlister Launcher invokes EDIF2NGD or XNF2NGD. If the input netlist is in another format that the Netlister Launcher recognizes, the Netlister Launcher invokes the program necessary to convert the netlist to EDIF or XNF format, then invokes EDIF2NGD or XNF2NGD. The netlist reader produces an NGO file for the top-level netlist file.

   If any subfiles are referenced in the top-level netlist (for example, a PAL description file, or another schematic file), the Netlister Launcher invokes the appropriate netlist reader for each of these files to convert each referenced file to an NGO file.

   The Netlister Launcher is described in the "Netlister Launcher" section. The netlist reader programs are described in the "EDIF2NGD" section and the "XNF2NGD" section.

2. Reduces all components in the design to NGD primitives

   To perform this step, NGDBuild merges components that reference other files by finding the referenced NGO files. NGDBuild also finds the appropriate system library components, physical macros (NMC files) and behavioral models.

3. Checks the design by running a Logical DRC (Design Rule Check) on the converted design

   The Logical DRC is a series of tests on the logical design. It is described in "Logical Design Rule Check" chapter.

4. Writes an NGD file as output

When NGDBuild reads the source netlist, it detects any files or parts of the design that have changed since the last run of NGDBuild. It updates files as follows:

- If you modified your input design, NGDBuild updates all of the files affected by the change and uses the updated files to produce a new NGD file.

  The Netlister Launcher checks timestamps (date and time information) for netlist files and intermediate NGDBuild files (NGOs). If an NGO file has a timestamp earlier than the netlist file that produced it, the NGO file is updated and a new NGD file is produced.

- NGDBuild completes the NGD production if all or some of the intermediate files already exist. These files may exist if you ran a netlist reader before you ran NGDBuild. NGDBuild uses the existing files and create the remaining files necessary to produce the output NGD file.

Note If the NGO for an netlist file is up to date, NGDBuild looks for an NCF file with the same base name as the netlist in the netlist directory and compares the timestamp of the NCF file against that of the NGO file. If the NCF file is newer, XNF2NGD or EDIF2NGD is run again. However, if an NCF file existed on a previous run of NGDBuild and the NCF file was deleted, NGDBuild does not detect that XNF2NGD or EDIF2NGD must be run again. In this case, you must use the –nt on option to force a rebuild. The –nt on option must also be used to force a rebuild if you change any of the XNF2NGD or EDIF2NGD options.

Syntax, files, and options for NGDBuild are described in the "NGDBuild" chapter.

Bus Matching

When NGDBuild encounters an instance of one netlist within another netlist, it requires that each pin specified on the upper-level instance match to a pin (or port) on the lower-level netlist. Two pins must have exactly the same name in order to be matched. This requirement applies to all FPGAs and CPLDs supported for NGDBuild.

If the interface between the two netlists uses bused pins, these pins are expanded into scalar pins before any pin matching occurs. For example, the pin A[7:0] might be expanded into 8 pins namedA[7] through A[0]. If both netlists use the same nomenclature (that is, the same index delimiter characters) when expanding the bused pin, the scalar pin names will match exactly. However, if the two netlists were created by different vendors and different delimiters are used, the resulting scalar pin names do not match exactly.

In cases where the scalar pin names do not match exactly, NGDBuild analyzes the pin names in both netlists and attempts to identify names that resulted from the expansion of bused pins. When it identifies a bus-expanded pin name, it tries several other bus-naming conventions to find a match in the other netlist so it can merge the two netlists. For example, if it finds a pin named A(3) in one netlist, it looks for pins named A(3), A[3], A<3> or A3 in the other netlist.

The following table lists the bus naming conventions understood by NGDBuild.

Table B-1 Bus Naming Conventions

Naming Convention	**Example**
busname(index)	DI(3)
busname<index>	DI<3>
busname[index]	DI[3]
busnameindex	DI3

If your third-party netlist writer allows you to specify the bus-naming convention, use one of the conventions shown in the preceding table to avoid "pin mismatch" errors during NGDBuild. If your third-party EDIF writer preserves bus pins using the EDIF "array" construct, the bus pins are expanded by EDIF2NGD using parentheses, which is one of the supported naming conventions.

Note NGDBuild support for bused pins is limited to this understanding of different naming conventions. It is not able to merge together two netlists if a bused pin has different indices between the two files. For example, it cannot match A[7:0] in one netlist to A[15:8] in another.

In the Xilinx UnifiedPro library for Virtex, some of the pins on the block RAM primitives are bused. If your third-party netlist writer uses one of the bus naming conventions listed in the preceding table or uses the EDIF array construct, these primitives are recognized properly by NGDBuild. The use of any other naming convention may result in an "unexpanded block" error during NGDBuild.

Netlister Launcher

The Netlister Launcher, which is part of NGDBuild, translates an EDIF or XNF netlist to an NGO file. NGDBuild uses this NGO file to create an NGD file.

Note The NGC netlist file does not require Netlister Launcher processing. It is equivalent to an NGO file. Also, it contains its own constraints information and cannot be processed with an NCF file.

When NGDBuild is invoked, the Netlister launcher goes through the following steps:

1. The Netlister Launcher initializes itself with a set of rules for determining what netlist reader to use with each type of netlist, and the options with which each reader is invoked.

   The rules are contained in the system rules file (described in the "System Rules File" section) and in the user rules file (described in the "User Rules File" section).

2. NGDBuild makes the directory of the top-level netlist the first entry in the Netlister Launcher's list of search paths.

3. For the top-level design and for each file referenced in the top-level design, NGDBuild queries the Netlist Launcher for the presence of the corresponding NGO file.

4. For each NGO file requested, the Netlister Launcher performs the following actions:

   ♦ Determines what netlist is the source for the requested NGO file

     The Netlister Launcher determines the source netlist by looking in its rules database for the list of legal netlist extensions. Then, it looks in the search path (which includes the current directory) for a netlist file possessing a legal extension and the same name as the requested NGO file.

   ♦ Finds the requested NGO file

     The Netlister Launcher looks first in the directory specified with the –dd option (or current directory if a directory is not specified). If the NGO file is not found there and the source netlist was not found in the search path, the Netlister Launcher looks for the NGO file in the search path.

- Determines whether the NGO file must be created or updated

  If neither the netlist source file nor the NGO file is found, NGDBuild exits with an error.

  If the netlist source file is found but the corresponding NGO file is not found, the Netlister Launcher invokes the proper netlist reader to create the NGO file.

  If the netlist source file is not found but the corresponding NGO file is found, the Netlister Launcher indicates to NGDBuild that the file exists and NGDBuild uses this NGO file.

  If both the netlist source file and the corresponding NGO file are found, the netlist file’s time stamp is checked against the NGO file’s timestamp. If the timestamp of the NGO file is later than the source netlist, the Netlister Launcher returns a “found” status to NGDBuild. If the timestamp of the NGO file is earlier than the netlist source, or the NGO file is not present in the expected location, then the Launcher creates the NGO file from the netlist source by invoking the netlist reader specified by its rules.

Note The timestamp check can be overridden by options on the NGDBuild command line. The –nt on option updates all existing NGO files, regardless of their timestamps. The –nt off option does not update any existing NGO files, regardless of their timestamps.

5. The Netlister launcher indicates to NGDBuild that the requested NGO files have been found, and NGDBuild can process all of these NGO files.

Netlister Launcher Rules Files

The behavior of the Netlister Launcher is determined by rules defined in the system rules file and the user rule file. These rules determine the following:

- What netlist source files are acceptable
- Which netlist reader reads each of these netlist files
- What the default options are for each netlist reader

The system rules file contains the default rules supplied with the Xilinx Development System software. The user rules file can add to or override the system rules.

User Rules File

The user rules file can add to or override the rules in the system rules file. You can specify the location of the user rules file with the NGDBuild –ur option. The user rules file must have a .urf extension. See the "–ur (Read User Rules File)" section of the "NGDBuild" chapter for more information.

User Rules and System Rules

User rules are treated as follows:

- A user rule can override a system rule if it specifies the same source and target files as the system rule.
- A user rule can supplement a system rule if its target file is identical to a system rule's source file, or if its source file is the same as a system rule's target file.
- A user rule that has a source file identical to a system rule's target file and a target file that is identical to the same system rule's source file is illegal, because it defines a loop.

User Rules Format

Each rule in the user rules file has the following format:

```
RuleName = <rulename1>;

<key1> = <value1>;

<key2> = <value2>;

  .

  .

  .

<keyn> = <valuen>;
```

Following are the keys allowed and the values expected:

Note The value types for the keys are described in the "Value Types in Key Statements" section.

- RuleName—This key identifies the beginning of a rule. It is also used in error messages relating to the rule. It expects a RULENAME value. A value is required.
- NetlistFile—This key specifies a netlist or class of netlists that the netlist reader takes as input. The extension of NetlistFile is used together with the TargetExtension to identify the rule. It expects either a FILENAME or an EXTENSION value. If a file name is specified, it should be just a file name (that is, no path). Any leading path is ignored. A value is required.
- TargetExtension—This key specifies the class of files generated by the netlist reader. It is used together with the extension from NetlistFile to identify the rule. It expects an EXTENSION value. A value is required.
- Netlister—This key specifies the netlist reader to use when translating a specific netlist or class of netlists to a target file. The specific netlist or class of netlists is specified by NetlistFile, and the class of target files is specified by TargetExtension. It expects an EXECUTABLE value. A value is required.

- NetlisterTopOptions—This key specifies options for the netlist reader when compiling the top-level design. It expects an OPTIONS value or the keyword NONE. Included in this string should be the keywords $INFILE and $OUTFILE, in which the input and output files is substituted. In addition, the following keywords may appear.
  - ♦ $PART—The part passed to NGDBuild by the –p option is substituted. It may include architecture, device, package and speed information. The syntax for a $PART specification is the same as described in the "–p (Part Number)" section of the "Introduction" chapter.
  - ♦ $FAMILY—The family passed to NGDBuild by the –p option is substituted. A value is optional.
  - ♦ $DEVICE—The device passed to NGDBuild by the –p option is substituted. A value is optional.
  - ♦ $PKG—The package passed to NGDBuild by the –p option is substituted. A value is optional.
  - ♦ $SPEED—The speed passed to NGDBuild by the –p option is substituted. A value is optional.
  - ♦ $LIBRARIES—The libraries passed to NGDBuild. A value is optional.
  - ♦ $IGNORE_LOCS—Substitute the –r option to EDIF2NGD or XNF2NGD if the NGDBuild command line contained a –r option.
  - ♦ $ADD_PADS—Substitute the –a option to EDIF2NGD if the NGDBuild command line contained a –a option.

  The options in the NetlisterTopOptions line must be enclosed in quotation marks.
- NetlisterOptions—This key specifies options for the netlist reader when compiling sub-designs. It expects an OPTIONS value or the keyword NONE. Included in this string should be the keywords $INFILE and $OUTFILE, in which the input and output files is substituted. In addition, any of the keywords that may be entered for the NetlisterTopOptions key may also be used for the NetlisterOptions key.

The options in the NetlisterOptions line must be enclosed in quotation marks.

- NetlisterDirectory—This key specifies the directory in which to run the netlist reader. The launcher changes to this directory before running the netlist reader. It expects a DIR value or the keywords $SOURCE, $OUTPUT, or NONE, where the path to the source netlist is substituted for $SOURCE, the directory specified with the -dd option is substituted for $OUTPUT, and the current working directory is substituted for NONE. A value is optional.
- NetlisterSuccessStatus—This key specifies the return code that the netlist reader returns if it ran successfully. It expects a NUMBER value or the keyword NONE. The number may be preceded with one of the following: =, <, >, or !. A value is optional.

Value Types in Key Statements

The value types used in the preceding key statements are the following:

- RULENAME—Any series of characters except for a semicolon “;” and white space (for example, space, tab, newline).
- EXTENSION—A “.” followed by an extension that conforms to the requirements of the platform.
- FILENAME—A file name that conforms to the requirements of the platform.
- EXECUTABLE—An executable name that conforms to the requirements of the platform. It may be a full path to an executable or just an executable name. If it is just a name, then the $PATH environment variable is used to locate the executable.
- DIR—A directory name that conforms to the requirements of the platform.
- OPTIONS—Any valid string of options for the executable.
- NUMBER—Any series of digits.
- STRING—Any series of characters in double quotes.

System Rules File

The system rules are shown following. The system rules file is not an ASCII file, but for the purpose of describing the rules, the rules are described using the same syntax as in the user rules file. This syntax is described in the "User Rules File" section.

Note If a rule attribute is not specified, it is assumed to have the value NONE.

```
#####################################################
# xnf2ngd rules
#####################################################

RuleName = XNF_RULE;
NetlistFile = .xnf;
TargetExtension = .ngo;
Netlister = xnf2ngd;
NetlisterTopOptions = "[-p $PART] -u [$IGNORE_LOCS] [$QUIET] {-l
$LIBRARIES}
$INFILE $OUTFILE";
NetlisterOptions  =  "[-p  $PART]  [$IGNORE_LOCS]  {-l  $LIBRARIES}
$INFILE $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = XTF_RULE;
NetlistFile = .xtf;
TargetExtension = .ngo;
Netlister = xnf2ngd;
NetlisterTopOptions = "[-p $PART] -u [$IGNORE_LOCS] [$QUIET] {-l
$LIBRARIES}
$INFILE $OUTFILE";
NetlisterOptions  =  "[-p  $PART]  [$IGNORE_LOCS]  {-l  $LIBRARIES}
$INFILE $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = XFF_RULE;
NetlistFile = .xff;
TargetExtension = .ngo;
Netlister = xnf2ngd;
NetlisterTopOptions = "[-p $PART] -u [$IGNORE_LOCS] [$QUIET] {-l
$LIBRARIES}
$INFILE $OUTFILE";
NetlisterOptions  =  "[-p  $PART]  [$IGNORE_LOCS]  {-l  $LIBRARIES}
$INFILE $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = XG_RULE;
NetlistFile = .xg;
TargetExtension = .ngo;
Netlister = xnf2ngd;
NetlisterTopOptions = "[-p $PART] -u [$IGNORE_LOCS] [$QUIET] {-l
$LIBRARIES}
$INFILE $OUTFILE";
NetlisterOptions  =  "[-p  $PART]  [$IGNORE_LOCS]  {-l  $LIBRARIES}
$INFILE $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = SYN_XNF_RULE;
NetlistFile = .sxnf;
TargetExtension = .ngo;
Netlister = xnf2ngd;
NetlisterTopOptions  =  "[-p  $PART]  -u  [$IGNORE_LOCS]  -l  synopsys
[$QUIET] {-l $LIBRARIES} $INFILE $OUTFILE";
NetlisterOptions  =  "[-p  $PART]  [$IGNORE_LOCS]  -l  synopsys  {-l
$LIBRARIES}
$INFILE $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

#####################################################
# edif2ngd rules
#####################################################

RuleName = EDN_RULE;
NetlistFile = .edn;
TargetExtension = .ngo;
Netlister = edif2ngd;
NetlisterTopOptions = "[$IGNORE_LOCS] [$ADD_PADS] [$QUIET] [$AUL]
{-l $LIBRARIES} $INFILE $OUTFILE";
```

```
NetlisterOptions = "-noa [$IGNORE_LOCS] {-l $LIBRARIES} $INFILE
$OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = EDF_RULE;
NetlistFile = .edf;
TargetExtension = .ngo;
Netlister = edif2ngd;
NetlisterTopOptions = "[$IGNORE_LOCS] [$ADD_PADS] [$QUIET] [$AUL]
{-l $LIBRARIES} $INFILE $OUTFILE";
NetlisterOptions = "-noa [$IGNORE_LOCS] {-l $LIBRARIES} $INFILE
$OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = EDIF_RULE;
NetlistFile = .edif;
TargetExtension = .ngo;
Netlister = edif2ngd;
NetlisterTopOptions = "[$IGNORE_LOCS] [$ADD_PADS] [$QUIET] [$AUL]
{-l $LIBRARIES} $INFILE $OUTFILE";
NetlisterOptions = "-noa [$IGNORE_LOCS] {-l $LIBRARIES} $INFILE
$OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

RuleName = SYN_EDIF_RULE;
NetlistFile = .sedif;
TargetExtension = .ngo;
Netlister = edif2ngd;
NetlisterTopOptions = NONE;
NetlisterOptions = "-l synopsys [$IGNORE_LOCS] {-l $LIBRARIES}
$INFILE $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;

#####################################################
# other rules
#####################################################

RuleName = PLD_RULE;
```

```
NetlistFile = .pld;
TargetExtension = .xnf;
Netlister = readpld;
NetlisterTopOptions = "-f $INFILE -t -ox $OUTFILE";
NetlisterOptions = "-f $INFILE -ox $OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;
```

Rules File Examples

The following sections provide examples of system and user rules. The first example is the basis for understanding the ensuing user rules examples.

Example 1: EDF_RULE System Rule

As shown in the "System Rules File" section, the EDF_RULE system rule is defined as follows.

```
RuleName = EDF_RULE;
NetlistFile = .edf;
TargetExtension = .ngo;
Netlister = edif2ngd;
NetlisterTopOptions = "[$IGNORE_LOCS] [$ADD_PADS] [$QUIET] [$AUL]
{-l $LIBRARIES} $INFILE $OUTFILE";
NetlisterOptions = "-noa [$IGNORE_LOCS] {-l $LIBRARIES} $INFILE
$OUTFILE";
NetlisterDirectory = NONE;
NetlisterSuccessStatus = 0;
```

The EDF_RULE instructs the Netlister Launcher to use EDIF2NGD to translate an EDIF file to an NGO file. If the top-level netlist is being translated, the options defined in NetlisterTopOptions are used; if a lower-level netlist is being processed, the options defined by NetlisterOptions are used. Because NetlisterDirectory is NONE, the Netlister Launcher runs EDIF2NGD in the current working directory (the one from which NGDBuild was launched). The launcher expects EDIF2NGD to issue a return code of 0 if it was successful; any other value is interpreted as failure.

Example 2: User Rule

Following is a another example of a User Rule:

```
// URF Example 2
RuleName = OTHER_RULE; // end-of-line comments are also allowed
NetlistFile = .oth;
TargetExtension = .edf;
Netlister = other2edf;
NetlisterOptions = "$INFILE $OUTFILE";
NetlisterSuccessStatus = 1;
```

The user rule OTHER_RULE defines a completely new translation, from a hypothetical OTH file to an EDIF file. To do this translation, the other2edf program is used. The options defined by NetlisterOptions are used for translating all OTH files, regardless of whether they are top-level or lower-level netlists (because no explicit NetlisterTopOptions is given). The launcher expects other2edf to issue a return code of 1 if it was successful; any other value be interpreted as failure.

After the Netlister Launcher uses OTHER_RULE to run other2edf and create an EDIF file, it uses the EDF_RULE system rule (shown in the preceding section) to translate the EDIF file to an NGO file.

Example 3: User Rule

Following is a another example of a User Rule:

```
// URF Example 3
RuleName = EDF_LIB_RULE;
NetlistFile = .edf;
TargetExtension = .ngo;
NetlisterOptions = "-l xilinxun $INFILE $OUTFILE";
```

Because both the NetlistFile and TargetExtension of this user rule match those of the system rule EDF_RULE (shown in the "Example 1: EDF_RULE System Rule" section), the EDF_LIB_RULE overrides the EDF_RULE system rule. Any settings that are not defined by the EDF_LIB_RULE are inherited from EDF_RULE. So EDF_LIB_RULE uses the same netlister (EDIF2NGD), the same top-level options, the same directory, and expects the same success status as EDF_RULE. However, when translating lower-level netlists, the options used are only "–l xilinxun $INFILE $OUTFILE." (There is no reason to use "–l xilinxun" on EDIF2NGD; this is for illustrative purposes only.)

Example 4: User Rule

Following is a another example of a User Rule:

```
// URF Example 4
RuleName = STATE_EDF_RULE;
NetlistFile = state.edf;
TargetExtension = .ngo;
Netlister = state2ngd;
```

Although the NetlistFile is a complete file name, this user rule also matches the system rule EDF_RULE (shown in the "Example 1: EDF_RULE System Rule" section), because the extensions of NetlistFile and TargetExtension match. When the Netlister Launcher tries to make a file called state.ngo, it uses this rule instead of the system rule EDF_RULE (assuming that state.edf exists). As with the previous example, the unspecified settings are inherited from the matching system rule. The only change is that the fictitious program state2ngd is used in place of EDIF2NGD.

Note that if EDF_LIB_RULE (from the example in the "Example 3: User Rule" section) and this rule were both in the user rules file, STATE_EDF_RULE includes the modifications made by EDF_LIB_RULE. So a lower-level state.edf is translated by running state2ngd with the "-l xilinxun" option.

NGDBuild File Names and Locations

Following are some notes about file names in NGDBuild:

- An intermediate file has the same root name as the design that produced it. An intermediate file is generated when more than one netlist reader is needed to translate a netlist to a NGO file.
- Netlist root file names in the search path must be unique. For example, if you have the design state.edn, you cannot have another design named state.xnf in any of the directories specified in the search path.
- NGDBuild and the Netlister Launcher support quoted file names. Quoted file names may have special characters (for example, a space) that are not normally allowed.
- If the output directory specified in the call to NGDBuild is not writable, an error is displayed and NGDBuild fails.

Glossary

Click on a letter, or scroll down to view the entire glossary.

A B C D E F G H I J L M N O P R S T U V W X

A

ABEL

Advanced Boolean Expression Lanaguage (ABEL) is a high-level language (HDL) and compilation system produced by Data I/O Corporation.

adder

An adder is a combinatorial circuit that computes the sum of two or more numbers.

address

An address is the identification of a storage location, such as a register or a memory cell.
checked for syntax errors.

architecture

Architecture is the common logic structure of a family of programmable integrated circuits. The same architecture can be realized in different manufacturing processes. Examples of Xilinx architectures are the XC4000 and XC9500 devices.

area constraints

Area constraints are created by the user or a process such as synthesis to direct the optimization process that takes place during design implementation.

ASIC

Application-specific integrated circuit (ASIC), is a full-custom circuit. In which every mask is defined by the customer or a semi-custom circuit (gate array) where only a few masks are defined.

attributes

Attributes are instructions placed on symbols or nets in an FPGA or CPLD schematic to indicate their placement, implementation, naming, directionality, or other properties.

In LogiBLOX, attributes are placed on the symbols, which generate the module instances. For example, the BOUNDS attribute determines the width of a bus and its indexes.

B

back-annotation

Back-annotation is the translation of a routed or fitted design to a timing simulation netlist.

behavioral design

Behavioral design is a technology-independent, text-based design that incorporates high-level functionality and high-level information flow.

behavioral design method

A behavioral design method defines a circuit in terms of a textual language rather than a schematic of interconnected symbols.

behavioral simulation

Also known as functional simulation. Behavioral simulation is usually performed on designs that are entered using a hardware definition language (HDL).

This type of simulation takes place during the pre-synthesis stage of HDL design. Functional simulation checks that the HDL code

describes the desired design behavior.
Behavioral simulation is a simulation process that is performed by interpreting the equations that define the design. The equations do not need to be converted to the logic that represents them.

binary

Binary is a numbering system based on base 2 with only two digits, 0 and 1.
Unsigned binary refers to non-negative binary representation.

bit

A bit is a a binary digit representing 0 or 1.

BIT file

A BIT file is the same as a bitstream file. See bitstream.

BitGen

Is a program that produces a bitstream for Xilinx device configuration. BitGen takes a fully routed NCD (Circuit Description) file as its input and produces a configuration bitstream, a binary file with a.bit extension.

bitstream

A bitstream is a stream of data that contains location information for logic on a device, that is, the placement of Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), (TBUFs), pins, and routing elements. The bitstream also includes empty placeholders that are filled with the logical states sent by the device during a readback. Only the memory elements, such as flip-flops, RAMs, and CLB outputs, are mapped to these placeholders, because their contents are likely to change from one state to another. When downloaded to a device, a bitstream configures the logic of a device and programs the device so that the states of that device can be read back.
A bitstream file has a .bit extension.

block

1. A block is a group of one or more logic functions.
2. A block is a schematic or symbol sheet. There are four types of blocks.
    - A Composite block indicates that the design is hierarchical.
    - A Module block is a symbol with no underlying schematic.
    - A Pin block represents a schematic pin.
    - An Annotate block is a symbol without electrical connectivity that is used only for documentation and graphics.

bonded

Bonded means connected by a wire.

boundary scan

Boundary scan is the method used for board-level testing of electronic assemblies. The primary objectives are the testing of chip I/O signals and the interconnections between ICs.
It is the method for observing and controlling all new chip I/O signals through a standard interface called a Test Access Port (TAP). The boundary scan architecture includes four dedicated I/O pins for control and is described in IEEE spec 1149.1.

buffer

A buffer is an element used to increase the current or drive of a weak signal and, consequently, increase the fanout of the signal. A storage element.

BUFT

A BUFT is a 3-state buffer.

bus

A group of two or more signals that carry closely-associated signals in an electronic design.

byte

A binary word consisting of eight bits. When used to store a number value, a byte can represent a number from 0 to 255.

C

CAE

Computer Aided Engineering. The original term for electronic design automation (EDA). Now, often refers to the sofware tools used to develop the manufacturing tooling for the production of electronic system such as for the panelization of circuit boards.

CAE tool

A Computer-Aided Engineering tool (CAE). Usually refers to programs such as Innoveda, Cadence, or Mentor Graphics that are used to perform design entry and design verification.

capacitance

Capacitance is the property that measures the storage of electrically separated charges.
It is also the load on a net.

carry logic

An architecture feature of the Xilinx XC4000 families. Carry logic is designed to speed-up and reduce the area of counters, adders, incrementers, decrementers, comparators, and subtractors. It is a special interconnect that speeds up the carry path of adders and counters from one CLB to another. This dedicated carry line runs along each column of CLBs as well as the top and bottom CLBs. FPGA Express can synthesize carry logic directly.

carry path

The carry path is the computation of the carries in addition or subtraction from one CLB to another.

cell

A cell is a hierarchical description of an FPGA device.

checksum

A checksum is a summation of bits or digits generated according to an arbitrary formula used for checking data integrity. To verify that the data represented by a checksum number has been entered correctly, verify that the checksum number generated after processing is the same as the initial number.

CLB

The Configurable Logic Block (CLB). Constitutes the basic FPGA cell. It includes two 16-bit function generators (F or G), one 8-bit function generator (H), two registers (flip-flops or latches), and reprogrammable routing controls (multiplexers).
CLBs are used to implement macros and other designed functions. They provide the physical support for an implemented and downloaded design. CLBs have inputs on each side, and this versatility makes them flexible for the mapping and partitioning of logic.

CCLK pin

The CCLK pin is the XChecker pin that provides the configuration clock for the device or devices during a download.

clock

A clock is a signal that represents the time that a wave stays at a High or Low state. The rising and falling edges of a clock square wave trigger the activity of the circuits.

clock buffer

A clock buffer is an element used to increase the current or drive of a weak clock signal and consequently increase its fanout.

clock enable

A clock enable is a binary signal that allows or disallows synchronous logic to change with a clock signal. When enabled, this control signal permits a device to be clocked and to become active. There are four different states. The two active High states are CE 0 disabled and CE 1 enabled. The two active Low states are $\overline{\text{CE}}$ 0 enabled and $\overline{\text{CE}}$ 1 disabled.

clock skew

Clock skew is the time differential between 2 or more destination pins in a path.

CMOS

Complementary Metal Oxide Semiconductor (CMOS). Is an advanced IC manufacturing process technology characterized by high integration, low cost, low power, and high performance.

combinatorial logic

Combinatorial logic refers to any primitives with the exception of storage elements such as flip-flops.

compiler

A compiler is a language interpreter. The Synopsys compiler interprets HDL and makes concurrent process implementations for target architectures.

component

A component is an instantiation or symbol reference from a library of logic elements that can be placed on a schematic.

configuration

Configuration is the process of loading design-specific bitstreams into one or more FPGA devices to define the functional operation of the logical blocks, their interconnections, and the chip I/O.

This concept also refers to the configuration of a design directory for a particular design library, such as the XC4000 library.

constraints

Constraints are specifications for the implementation process. There are several categories of constraints: routing, timing, area, mapping, and placement constraints.

Using attributes, you can force the placement of logic (macros) in CLBs, the location of CLBs on the chip, and the maximum delay between flip-flops. PAR does not attempt to change the location of constrained logic.

constraints file

A constraints file specifies constraints (location and path delay) information in a textual form. An alternate method is to place constraints on a schematic.

contention

Contention is the state in which multiple conflicting outputs drive the same net.

counter

A counter is a circuit, composed of registers, that counts pulses, often reacting or causing a reaction to a predetermined pulse or series of pulses. Also called a divider, sometimes accumulator.

CPLD

Complex Programmable Logic Device (CPLD). Is an erasable programmable logic device that can be programmed with a schematic or a behavioral design. CPLDs constitute a type of complex PLD based on EPROM or EEPROM technology. They are characterized by an architecture offering high speed, predictable timing, and simple software.

The basic CPLD cell is called a macrocell, which is the CPLD implementation of a CLB. It is composed of AND gate arrays and is surrounded by the interconnect area.

CPLDs consume more power than FPGA devices, are based on a different architecture, and are primarily used to support behavioral designs and to implement complex counters, complex state machines, arithmetic operations, wide inputs, and PAL crunchers.

D

daisy chain

A daisy chain is a series of bitstream files concatenated in one file. It can be used to program several FPGAs connected in a daisy chain board configuration.

dangling bus

A dangling bus connects to a component pin or net at one end and unconnects at the other. A small filled box at the end of the bus indicates a dangling bus.

dangling net

A dangling net connects to a component pin or net at one end and unconnects at the other. A small filled box at the end of the net indicates a dangling net.

debugging

Debugging is the process of reading back or probing the states of a configured device to ensure that the device is behaving as expected while in circuit.

decimal

Decimal refers to a numbering system with a base of 10 digits starting with zero.

decoder

A decoder is a symbol that translates *n* input lines of binary information into 2*n* output lines. It is the opposite of an encoder.

Delay Locked Loop (DLL)

A digital circuit used to perform clock management functions on and off-chip.

density

Density is the number of gates on a device.

design implementation

Design implementation is a design implementation specification as opposed to the functional specification of the design. The implementation specification refers to the actual implementation of the design from low-level components expressed in bits. The functional specification refers to the definition of the design or circuit function.

The two common implementation tools are module generators (LogiBLOX or LPM) and synthesis packages.

device

A device is an integrated circuit or other solid-state circuit formed in semiconducting materials during manufacturing.

digital

Digital refers to the representation of information by code of discrete elements, as opposed to the continuous scale of analog representation.

DONE (XC4000) pin

The DONE pin is a dual-function pin. As an input, it can be configured to delay the global logic initialization or the enabling of outputs. As an output, it indicates the completion of the configuration process.

downloading

Downloading is the process of configuring or programming a device by sending bitstream data to the device.

DRC

The Design Rule Checker (DRC). A program that checks the (NCD) file for design implementations for errors.

DSP

Digital Signal Processing (DSP). A powerful and flexible technique of processing analog (linear) signals in digital form used in CoreGen.

E

EDA

Electronic Design Automation (EDA). A generic name for all methods of entering and processing digital and analog designs for further processing, simulation, and implementation.

edge decoder

An edge decoder is a decoder whose placement is constrained to precise positions within a side of the FPGA device.

EDIF

EDIF is the Electronic Data Interchange Format, an industry standard file format for specifying a design netlist. It is generated by a third-party design-entry tool. In the Xilinx M1 flow, EDIF is the standard input format.

effort level

Effort level refers to how hard the Xilinx Design System (XDS) tries to place a design. The effort level settings are.

- High, which provides the highest quality placement but requires the longest execution time. Use high effort on designs that do not route or do not meet your performance requirements.
- Medium, which is the default effort level. It provides the best trade-off between execution time and high quality placement for most designs.
- Low, which provides the fastest execution time and adequate placement results for prototyping of simple, easy-to-route designs. Low effort is useful if you are exploring a large design space and only need estimates of final performance.

ENRead

Mentor Graphics EDIF netlist reader. Translates an EDIF netlist into an EDDM single object.

entity

An entity is a set of interconnected components.

EPROM

An EPROM is an erasable PROM, which can be reprogrammed many times. Previous programs are simply erased by exposing the chip to ultra-violet light.
An EEPROM, or electrically erasable PROM, is another variety of EPROM that can be erased electrically.

F

FD

FD is a D flip-flop used in CLBs. Contrast with IFD.

FDSD

FDSD is a D flip-flop with Set Direct.

FIFO

A FIFO is a serial-in / serial-out shift register.

fitting

Fitting is the process of putting logic from your design into physical macrocell locations in the CPLD. Routing is performed automatically, and because of the UIM architecture, all designs are routable.

fitter

The fitter is the software that maps a PLD logic description into the target CPLD.

flat design

A flat design is a design composed of multiple sheets at the top-level schematic.

flattening

Flattening is the process of resolving all of the hierarchy references in a design. If a design contains several instantiations of a logic module, the flattened version of that design will duplicate the logic for each instantiation. A flattened design still contains hierarchical names for instances and nets.

flip-flop

A flip-flop is a simple two-state logic buffer activated by a clock and fed by a single input working in combination with the clock. The states are High and Low. When the clock goes High, the flip-flop works as a buffer as it outputs the value of the D input at the time the clock rises. The value is kept until the next clock cycle (rising clock edge). The output is not affected when the clock goes Low (falling clock edge).

floorplanning

Floorplanning is the process of choosing the best grouping and connectivity of logic in a design.
It is also the process of manually placing blocks of logic in an FPGA where the goal is to increase density, routability, or performance.

flow

The flow is an ordered sequence of processes that are executed to produce an implementation of a design.

FMAP

An FMAP is a symbol that defines mapping into a 4-input function generator (F or G).

FPGA

Field Programmable Gate Array (FPGA), is a class of integrated circuits pioneered by Xilinx in which the logic function is defined by the customer using Xilinx development system software after the IC has been manufactured and delivered to the end user. Gate arrays are another type of IC whose logic is defined during the manufacturing process. Xilinx supplies RAM-based FPGA devices.
FPGA applications include fast counters, fast pipelined designs, register intensive designs, and battery powered multi-level logic.

function generator

A function generator is a look-up table or black box with three or four inputs implementing any combinational functions of $(2^2)^4$ or 256 functions or $(2^2)^2$ or 65556 functions. The output is any value resulting from the logical functions executed within the box. The function generator implements a complete truth table, allowing speedy prediction of the output.

functional simulation

Functional simulation is the process of identifying logic errors in your design before it is implemented in a Xilinx device. Because timing information for the design is not available, the simulator tests the logic in the design using unit delays. Functional simulation is usually performed at the early stages of the design process.

G

gate

A gate is an integrated circuit composed of several transistors and capable of representing any primitive logic state, such as AND, OR, XOR, or NOT inversion conditions. Gates are also called digital, switching, or logic circuits.

gate array

A gate array is part of the ASIC chip. A gate array represents a certain type of gate repeated all over a VLSI-type chip. This type of logic requires the use of masks to program the connections between the blocks of gates.

global buffers

Global buffers are low-skew, high-speed buffers that connect to long lines. They do not map logic.

There is one BUFGP and one BUFGS in each corner of the chip. Primary buffers must be driven by an IOB. Secondary buffers can be driven by internal logic or IOBs.

global Set/Reset net

A global Set/Reset net is a high-speed, no-skew dedicated net, which reduces delays and routing congestion. This net accesses all flip-flops on the chip and can reinitialize all CLBs and IOBs.

global 3-state net

A global 3-state net is a net that forces all device outputs to high-impedance state unless boundary scan is enabled and executes an EXTEST instruction.

GND pin

The GND pin is Ground (0 volts).

group

A group is a collection of common signals to form a bus. In the case of a counter, for example, the different signals that produce the actual counter values can be combined to form an alias, or group.

guide file

A guide file is a previously placed and routed NCP file that can be used in a subsequent place and route operation.

guided design

Guided design is the use of a previously implemented version of a file for design mapping, placement, and routing. Guided design allows logic to be modified or added to a design while preserving the layout and performance that have been previously achieved.

H

HDL

Hardware Description Language. A language that describes circuits in textual code. The two most widely accepted HDLs are VHDL and Verilog.
An HDL, or hardware description language, describes designs in a technology-independent manner using a high level of abstraction. The most common HDLs in use today are Verilog and VHDL.

hexadecimal

Hexadecimal is a numbering system with a base of 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

hierarchical design

A hierarchical design is a design composed of multiple sheets at different levels of your schematic.

HMAP

An HMAP is a symbol that defines mapping into a three-input function generator (H). The HMAP symbol has two FMAP inputs and, optionally, one non-FMAP input.

hold time

Hold time is the time following a clock event during which the data input to a latch or flip-flop must remain stable in order to guarantee that the latched data is correct.

I

IBUF

An IBUF acts as a protection for the chip, shielding it from eventual current overflows.

IC

Integrated Circuit (IC) is a single piece of silicon on which thousands or millions of transistors are combined. ICs are the major building blocks of modern electronic systems.

IEEE

Institute of Electrical and Electronics Engineers.Pronounced I triple E.

IFD

IFD is an IOB flip-flop.

impedance

Impedance is the sum of all resistance and reactance of a circuit to the flow of alternating current.

implementation

Implementation is the mapping, placement and routing of a design. A phase in the design process during which the design is placed and routed.

incremental design

Incremental design refers to the implementation and verification of a design in stages using guided design.

indexes

Indexes are the left-most and right-most bits of a bus defining the bus range and precision.

input

An input is the symbol port through which data is sourced.

input pad registers and latches

Input pad registers and latches are D-type registers located in the I/O pad sections of the device. Input pad registers can be used instead of macrocell resources.

instance

An instance is one specific gate or hierarchical element in a design or netlist. The term "symbol" often describes instances in a schematic drawing. Instances are interconnected by pins and nets. Pins are ports through which connections are made from an instance to a net. A design that is flattened to its lowest level constituents is described with primitive instances.

instantiation

Instantiation is the act of placing a symbol that represents a primitive or a macro in a design or netlist.

interconnect

Interconnect is the metal in a device that is used to implement the nets of the design.

interconnect line

An interconnect line is any portion of a net.

IOB (input/output block)

An IOB is a collection or grouping of basic elements that implement the input and output functions of an FPGA device.

I/O pads

I/O pads are the input/output pads that interface the design logic with the pins of the device.

J

JEDEC

JEDEC is a CPLD file format used for downloading device bitmap information to a device programmer.

L

latch

A latch is a two-state buffer fed by two inputs, D and L. When the L input is Low, it acts as a transparent input; in this case, the latch acts as a buffer and outputs the value input by D. When the L input is High, it ignores the D input value.

library

A library is a set of macros, such as adders, buffers, and flip-flops that is part of the Xilinx interface. A library is used to create schematic designs.

load

A load is an input port.

logic

Logic is one of the three major classes of ICs in most digital electronic systems: microprocessors, memory, and logic. Logic is used for data manipulation and control functions that require higher speed than a microprocessor can provide.

logic allocation file

A logic allocation file, design.ll file, designates a file used for probing. The file provides bit locations of the values of RAM, I/O, latches, and flip-flops.

logic optimization

Logic optimization is the process that decreases the area or increases the speed of a design.

longline

A longline connects to a primary global net or to any secondary global net. Each CLB has four dedicated vertical longlines. These lines are very fast.

look-up table (LUT)

A Look-Up Table (LUT), implements Boolean functions.
See function generator.

low

Low is a logical state for which no output is generated.

LSB

An LSB, or least significant bit, is the left-most bit of the bus bounds or indexes. In one-hot and twos-complement encoding, the LSB is the right-most bit.

M

macro

A macro is a component made of nets and primitives, flip-flops, or latches that implements high-level functions, such as adders, subtracters, and dividers. Soft macros and Relationally Placed Macros (RPMs) are types of macros.

macrocell

A macrocell is the CPLD logic cell, which is made of gates only. A macrocell can implement both combinational and registered equations. High-density function block macrocells also contain an arithmetic logic unit (ALU) for implementing arithmetic functions.

mapping

Mapping is the process of assigning a design’s logic elements to the specific physical elements that actually implement logic functions in a device.

MCS-86 (Intel)

MCS-86 (Intel) is a PROM format supported by the Xilinx tools. Its maximum address is 1 048 576. This format supports PROM files of up to (8 x 1 048 576) = 8 388 608 bits.

microprocessor

A silicon chip that contains a CPU. Microprocessors control the logic of almost all digital devices, e.g. PCs, workstations, clock radios, and fuel-injection systems for automobiles

MSB

The Most Significant Bit (MSB) is the right-most bit of the bus bounds or indexes. In one-hot binary and twos-complement encoding, the MSB is the left-most bit.

MultiLINX

A cable designed to function as a download, read back, verification and logic probing tool for the larger Xilinx devices. MultiLINX functions as a USB device to send and receive data from host.

multiplexer

A multiplexer is a reprogrammable routing control. This component selects one input wire as output from a selection of wires.

N

net

1. A logical connection between two or more symbol instance pins. After routing, the abstract concept of a net is transformed to a physical connection called a wire.
2. An electrical connection between components or nets. It can also be a connection from a single component. It is the same as a wire or a signal.

netlist

A netlist is a text description of the circuit connectivity. It is basically a list of connectors, a list of instances, and, for each instance, a list of the signals connected to the instance terminals. In addition, the netlist contains attribute information.

network

A network is a collection of logic elements and the wires (nets or connections) that define how they interconnect.

O

optimization

Optimization is the process that decreases the area or increases the speed of a design.

oscillator

An oscillator is a bi-stable circuit that can be used as a clock. The stable states are 0 and 1.

P

package

A package is the physical packaging of a chip, for example, PG84, VQ100, and PC48.

pad

A pad is the physical bonding pad on an integrated circuit. All signals on a chip must enter and leave by way of a pad. Pads are connected to package pins in order for signals to enter or leave an integrated circuit package.

PAL

A PAL is a programmable logic device that consists of a programmable AND matrix whose outputs drive fixed OR gates. This was one of the earliest forms of programmable logic. PALs can typically implement small functions easily (up to a hundred gates) and run very fast, but they are inefficient for large functions.

Parallel Cable III

Parallel Cable III is a cable assembly which contains a buffer to protect your PCs parallel port and a set of headers to connect to your target system.

path

A path is a connected series of nets and logic elements. A path has a start point and an end point that are different depending on the type of path. The time taken for a signal to propagate through a path is referred to as the path delay.

path delay

Path delay is the time it takes for a signal to propagate through a path.

period

The period is the number of steps in a clock pattern multiplied by the step size.

pin

A pin can be a symbol pin or a package pin. A package pin is a physical connector on an integrated circuit package that carries signals into and out of an integrated circuit. A symbol pin, also referred to as an instance pin, is the connection point of an instance to a net.

PIP (programmable interconnect points)

Programmable interconnect points, or PIP, provide the routing paths used to connect the inputs and outputs of IOBs and CLBs into logic networks.
A PIP is made of a CMOS transistor, which you can turn on and off to activate the PIP.

placing

Placing is the process of assigning physical device cell locations to the logic in a design.

PLD

A Programmable Logic Device (PLD), is composed of two types of gate arrays: the AND array and the OR array, thus providing for sum of products algorithmic representations. PLDs include three distinct types of chips: PROMs, PALs, and PLAs. The most flexible device is the PLA (programmable logic array) in which both the AND and OR gate arrays are programmable. In the PROM device, only the OR gate array is programmable. In the PAL device, only the AND gate array is programmable. PLDs are programmed by blowing the fuses along the paths that must be disconnected.
FPGAs and CPLDs are classes of PLDs.

post-synthesis simulation

This type of simulation is usually done after the HDL code has been expanded into gates. Post-synthesis simulation is similar to behavioral simualtion since design behavior is being checked. The difference is that in post-synthesis simulation the synthesis tool’s results are being checked. If post-synthesis and behavioral simulation match, then the HDL synthesis tool has interpreted the HDL code correctly.

primitive

1. A basic logic element, such as a gate (AND, OR, XOR, NAND, or NOR), inverter, flip-flop, or latch.
2. A logic element that directly corresponds, or maps, to one of these basic elements.

programming

Programming is the process of configuring the programmable interconnect in the FPGA.

PROM

A PROM is a programmable read-only memory.

PROM file

A PROM file consists of one or more BIT files (bitstreams) formed into one or more datastreams. The file is formatted in one of three industry-standard formats: Intel MCS86 HEX, Tektronics TEKHEX, or Motorola EXORmacs. The PROM file includes headers that specify the length of the bitstreams as well as all the framing and control information necessary to configure the FPGAs. It can be used to program one or more devices.

propagation

Propagation is the repetition of the bus attributes in LogiBLOX along a data path so that they only need to be defined on one bus in a data path.

pull-down resistor

A pull-down resistor is a device or circuit used to reduce the output impedance of a device, often a resistor network that holds a device or circuit output at or less than the zero input level of a subsequent digital device in a system.

pull-up resistor

A pull-up resistor is a device or method used to keep the output voltage of a device at a high level, often a resistor network connected to a positive supply voltage.

R

RAM

Random Access Memory (RAM) is a read / write memory that has an access time independent of the physical location of the data.
RAM can be used to change the address values (16^1) of the function generator it is a part of.

readback

Readback is the process of reading the logic downloaded to an FPGA device back to the source. There are two types of readback.

1. A readback of logic usually accompanied by a comparison check to verify that the design was downloaded in its entirety.
2. A readback of the states stored in the device memory elements to ensure that the device is behaving as expected.

register

A register is a set of flip-flops used to store data. It is an accumulator used for all arithmetic operations.

resistance

1. The property — based on material, dimensions, and temperature of conductors — that determines the amount of current produced at a given difference in potential. A material's current impedance that dissipates power in the form of heat.
2. The drive of the output pins on a network.

resistor

A resistor is a device that provides resistance.

ROM

Read Only Memory (ROM) is a static memory structure that retains a state indefinitely, even when the power is turned off. It can be part of a function generator.

routing

Routing is the process of assigning logical nets to physical wire segments in the FPGA that interconnect logic cells.

RPM

A Relationally Placed Macro (RPM) defines the spatial relationship of the primitives that constitute its logic. An indivisible block of logic elements that are placed as a unit into a design.

RTL

Resistor Transistor Logic

S

schematic

A schematic is a hierarchical drawing representing a design in terms of user and library components.

script

A script is a series of commands that automatically execute a complex operation such as the steps in a design flow.

seed

A seed is a random number that determines the order of the cells in the design to be placed.

set/reset

This operation is made possible by the asynchronous set/reset property. This function is also implemented by the Global Reset STARTUP primitive.

shift register

A shift register is a register in which data is loaded in parallel and shifted out of the register again. It refers to a chain of flip-flops connected in cascade.

signal

A signal is a wire or a net. See “net.”

simulation

Simulation is the process of verifying the logic and timing of a design.

skew

Skew is clock delay. See clock skew.

slew rate

The slew rate is the speed with which the output voltage level transitions from +5 V to 0 V or vice-versa. The slew rate determines how fast the transistors on the outputs change states.

slice

Two slices form a CLB within Virtex and Spartan-II families.

speed

Speed is a function of net types, CLB density, switching matrices, and architecture.

STARTUP symbol

The STARTUP symbol is a symbol used to set/reset all CLB and IOB flip-flops.

state

A state is the set of values stored in the memory elements of a device (flip-flops, RAMs, CLB outputs, and IOBs) that represent the state of that device at a particular point of the readback cycle. To each state there corresponds a specific set of logical values.

state machine

A state machine is a set of combinatorial and sequential logic elements arranged to operate in a predefined sequence in response to specified inputs. The hardware implementation of a state machine design is a set of storage registers (flip-flops) and combinatorial logic, or gates. The storage registers store the current state, and the logic network performs the operations to determine the next state.

static timing analysis

A static timing analysis is a point-to-point delay analysis of a design network.

T

TEKHEX (Tektronix)

TEKHEX (Tektronix) is a PROM format supported by Xilinx. Its maximum address is 65 536. This format supports PROM files of up to (8 x 65 536) = 524 288 bits.

testbench

An HDL netlist containing test vectors to drive a simulation.

threshold

The threshold is the crossover point when something occurs or is observed or indicated. The CMOS threshold and TTL threshold are examples.

timing

Timing is the process that calculates the delays associated with each of the routed nets in the design.

timing simulation

This type of simulation takes place after the HDL design has been synthesized and placed and routed. The purpose of this simulation is to check the dynamic timing behavior of the HDL design in the target technology.
Use the block and routing delay information from the routed design to assess the circuit behavior under worst-case conditions.

timing specifications

Timing specifications define the maximum allowable delay on any given set of paths in a design. Timing specifications are entered on the schematic.

transistor

A transistor is a three-terminal semiconductor device that switches or amplifies electrical current. It acts like a switch: On is equal to 1, and Off is equal to 0.

trimming

Trimming is the process of removing unconnected or unused logic.

tristate (3-state)

A 3-state, or 3-state buffer, is a buffer that places an output signal in a high-impedance state to prevent it from contending with another output signal.

tristate (3-state) condition

A 3-state condition is a high-impedance state. A 3-state can act also as a normal output; i.e. it can be on, off, or not connected.

truth table

A truth table defines the behavior for a block of digital logic. Each line of a truth table lists the input signal values and the resulting output value.

TTL

TTL, or transistor-transistor logic, is a technology with specific interchange (communication of digital signals) voltages and currents. Other technologies include ECL, MOS, and CMOS. These types of logic are used as criteria to classify digital integrated circuits.

U

unbonded

Unbonded describes an IOB used for internal logic only. This element does not have an external package pin.

Unified Libraries

The Unified Libraries are a set of logic macros and functions that are used to define the logic of a design. The elements are compatible across families and schematic editors. For example, a Mentor Graphics symbol has the same configuration as a Innoveda symbol; that is, it has the same footprint and name. On the other hand, the Unified Libraries support device-independent design, allowing a design to be retargeted to different devices with minimal overhead; thus, a Innoveda XC2000 macro will be similar to an XC3000 macro. Unified Library Xilinx library standard which emphasizes standardization of component naming and physical appearance of all schematic symbols across all FPGA and CPLD architectures.

V

VCC pin

The VCC pin is Power (5 volts). It is the supply voltage.

verification

Verification is the process of reading back the configuration data of a device and comparing it to the original design to ensure that all of the design was correctly received by the device.

Verilog

Verilog is a commonly used Hardware Description Language (HDL) that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. It is IEEE standard 1364-1995. Verilog was originally developed by Cadence Design Systems and is now maintained by OVI.
A Verilog file has a .v extension.

VHDL

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC an acronym for Very High-Speed Integrated Circuits). It can be used to describe the concurrent and sequential behavior of a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. VHDL is IEEE standard 1076-1993.
A VHDL file has a .vhd or .vhdl extension.

VITAL

VITAL is an acronym for VHDL Intitiative Toward ASIC Libraries. It is a VHDL-library standard (IEEE 1076.4) that defines standard constructs for simulation modeling, accelerating, and improving the performance of VHDL simulators.

W

wire

A wire is a net or a signal. See net.

X

XFF file

An XFF file is a flattened XNF file, including all the XNF files that are part of a design. XNFMerge generates this file.

Index