



The Fastest Verification

ZeBu-Server™ Compilation Manual

Document revision – c –

December 2010

Version 6_3_0

Copyright © 2009-2010 EVE. All rights reserved.

This publication is confidential and may not be reproduced, in whole or in part,
in any manner or in any form, without prior written permission of EVE.



Copyright Notice Proprietary Information

Copyright © 2009-2010 EVE. All rights reserved.

This software and documentation contain confidential and proprietary information that is the property of EVE. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of EVE, or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with EVE permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of EVE, for the exclusive use of _____ and its employees. This is copy number ____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

EVE AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.



Table of Contents

ABOUT THIS MANUAL	8
OVERVIEW	8
HISTORY	9
RELATED DOCUMENTATION	10
1 INTRODUCTION	11
1.1 ZEBU-SERVER HARDWARE OVERVIEW	11
1.2 COMPILATION FLOW	13
1.3 COMPILATION METHODOLOGY	14
2 QUICK GUIDE FOR zCUI INTERFACE	15
2.1 zCUI OPENING VIEW	15
2.2 DESCRIPTION OF THE zCUI VIEWS	16
2.2.1 <i>Project View</i>	16
2.2.2 <i>Menu bar</i>	19
2.2.3 <i>Toolbars</i>	19
2.3 COMPILATION VIEW	20
2.3.1 <i>Search Tools for Source files and Logs</i>	21
2.3.2 <i>Selection in a Log for Copy-Paste</i>	21
3 PREPARING THE COMPILATION PROJECT	22
3.1 INTRODUCTION	22
3.1.1 <i>Inputs of the ZeBu Compiler</i>	22
3.1.2 <i>Organizing the Input Files</i>	22
3.2 RTL SOURCE FILES	23
3.2.1 <i>Choosing the synthesizer</i>	23
3.2.2 <i>Supported RTL Source Files</i>	24
3.2.3 <i>Declaring the RTL source files</i>	24
3.2.4 <i>Defining the Synthesis Options</i>	27
3.3 EDIF SOURCE FILES	33
3.3.1 <i>Declaring EDIF Source Files in zCui</i>	33
3.3.2 <i>Integrating DesignWare Library Components with ZeBu</i>	34
3.4 VERIFICATION ENVIRONMENT	35
3.4.1 <i>Design Verification Environment (DVE) file</i>	35
3.4.2 <i>Hardware and Software Co-Simulation</i>	37
3.4.3 <i>Integrating a Synthesizable Testbench</i>	38
3.4.4 <i>Connecting to a Software Debugger</i>	38
3.4.5 <i>Integrating a Transactor from EVE Vertical Solutions' Catalog</i>	39
3.4.6 <i>Integrating a ZEMI-3 Transactor</i>	39
3.4.7 <i>SRAM Trace</i>	40
3.4.8 <i>Mixed Environments</i>	41
3.4.9 <i>Example of DVE File</i>	41
3.4.10 <i>Transactor Mapping</i>	44



3.5	CLOCK MODELING.....	45
3.5.1	Definitions.....	45
3.5.2	Declaring clocks	46
3.5.3	Clock Modeling for Primary Clocks.....	47
3.5.4	Clock Modeling for Derived Clocks	48
3.5.5	Static Timing Analysis	50
3.6	DISTRIBUTING THE DESIGN IN THE SYSTEM	51
3.6.1	Introduction	51
3.6.2	System-level Partitioning	52
3.6.3	zCore-level Partitioning.....	55
3.6.4	Modifying the settings in the Clustering panel	57
3.7	SPECIFIC DESIGN MODELING FOR ZEBU	60
3.7.1	Introduction	60
3.7.2	RTL-based Compilation Scripts	61
3.7.3	Handling Initializations.....	62
3.7.4	Handling tristates	63
3.7.5	Forcing Nets in the Design	66
3.7.6	Declaring Signals for Dynamic Control at Runtime (Dynamic Force)	70
3.7.7	Processing Design Blackboxes	73
3.8	MEMORY MODELING	76
3.8.1	Introduction	76
3.8.2	Implementation Guidelines.....	76
3.8.3	Automatic Memory Modeling when synthesizing with zFAST	77
3.8.4	Inputs of the ZeBu Memory Generator.....	78
3.8.5	Settings for automatic selection of the memory physical resources	79
3.8.6	Writing a Memory Source file	80
3.9	DEBUG	88
3.9.1	Accessing Signals in ZeBu	88
3.9.2	Declaring the Probes in a Tcl File.....	90
3.9.3	SystemVerilog Assertions (SVAs) with zFAST	92
3.9.4	Summary of the Compilation Settings for Debug.....	92
3.9.5	Detection of Combinational Loops	93
4	COMPILING WITH zCUI	95
4.1	COMPILATION SETTINGS.....	95
4.1.1	Declaring the Hardware Configuration File.....	95
4.1.2	Options for storage of FPGA P&R intermediate files.....	95
4.1.3	Configuring the Compiler for Job Scheduling.....	96
4.1.4	Retry mechanism for NFS failures.....	98
4.1.5	Compiling a design for relocation	99
4.2	COMPILING THE PROJECT	99
4.2.1	Choosing the compilation target	99
4.2.2	Launching the Compilation.....	100
4.3	MONITORING THE COMPILATION	101
4.3.1	During Compilation.....	101
4.3.2	Successful Compilation	102



4.4	OTHER OPTIONS	103
4.4.1	<i>Changing the editor for source files and logs</i>	<i>103</i>
4.5	zCUI ARCHIVER FEATURE.....	103
4.5.1	<i>Archival in Batch Mode</i>	<i>103</i>
4.5.2	<i>Archival in Graphical Mode.....</i>	<i>105</i>
5	ADVANCED COMPILATION.....	107
5.1	RETARGETING FROM AN ASIC SYNTHESIZER.....	107
5.1.1	<i>Retargeting a Design to GTECH.....</i>	<i>107</i>
5.1.2	<i>Flattening the Netlist</i>	<i>108</i>
5.1.3	<i>Creating a Dedicated Translation Library.....</i>	<i>108</i>
5.2	ADVANCED CLOCK MODELING	110
5.2.1	<i>ZeBu-Server Uncontrolled Clocks</i>	<i>110</i>
5.2.2	<i>Configuring the allocation of clock resources for user clocks.....</i>	<i>111</i>
5.3	COMPILING TO OPTIMIZE RUNTIME PERFORMANCE	112
5.3.1	<i>Timing-driven Optimization</i>	<i>112</i>
5.3.2	<i>Reduction of Inter-FPGA Clocks</i>	<i>112</i>
5.4	FPGA PLACE AND ROUTE.....	113
5.4.1	<i>Introduction</i>	<i>113</i>
5.4.2	<i>Parallel Automatic Recompilation of Failing FPGAs (PARFF).....</i>	<i>113</i>
5.4.3	<i>Manual Modification of FPGA Place & Route Options.....</i>	<i>116</i>
5.5	INFORMATION FOR ADDITIONAL COMMAND FILES	119
5.5.1	<i>Declaring commands for the zCore-level Compiler</i>	<i>119</i>
5.5.2	<i>Automatic zCore Generation</i>	<i>120</i>
5.5.3	<i>Advanced Commands for Performance Oriented Partitioning</i>	<i>121</i>
6	ZEBU-SERVER DOCUMENT PACKAGE.....	122
7	EVE CONTACTS.....	123



Figures

Figure 1: ZeBu-Server 2-slot and 5-slot Units	11
Figure 2: Architecture of an FPGA module.....	12
Figure 3: ZeBu-Server Runtime Architecture.....	12
Figure 4: Compilation Flow	13
Figure 5: Opening display (<i>Project View</i>)	15
Figure 6: <i>Compilation View</i>	15
Figure 7: Details of the <i>Project View</i> with a <i>Properties</i> Pane.....	16
Figure 8: Details of the <i>Project View</i> for a Source File (<i>Inspector</i> pane)	18
Figure 9: Additional Elements in zCui GUI.....	18
Figure 10: zCui Menus.....	19
Figure 11: Toolbar Contextual Menu.....	19
Figure 12: zCui Toolbars	19
Figure 13: <i>Compilation View</i>	20
Figure 14: Search Toolbar.....	21
Figure 15: <i>RTL Group Properties</i> panel for the Default RTL Group	24
Figure 16: Contextual menu for an RTL group.....	25
Figure 17: Contextual menu for an RTL source file.....	25
Figure 18: <i>RTL Group Properties</i> panel and RTL source files added.....	26
Figure 19: <i>RTL Group Properties</i> → <i>Main</i> tab for zFAST.....	27
Figure 20: <i>RTL Group Properties</i> → zFAST tab	28
Figure 21: <i>RTL Group Properties</i> → <i>Main</i> tab for third-party synthesizers	30
Figure 22: <i>RTL Group Properties</i> → <i>Synthesizer</i> tab.....	31
Figure 23: <i>RTL Group Properties</i> → <i>Blackboxes</i> tab	32
Figure 24: <i>RTL Group Properties</i> → <i>Clocks</i> tab	33
Figure 25: Contextual Menu to add EDIF Source Files.....	33
Figure 26: Contextual Menu for an EDIF source file.....	33
Figure 27: <i>Environment</i> → <i>DVE</i> tab for Automatic Generation	37
Figure 28: <i>Environment</i> → <i>External Transactors</i> tab for an EVE Transactor	39
Figure 29: Creating a ZEMI-3 Transactor.....	39
Figure 30: Adding Source Files for a ZEMI-3 Transactor	40
Figure 31: System Architecture for DVE Example	42
Figure 32: <i>Environment</i> → <i>Transactor Mapping</i> tab.....	44
Figure 33: Design Clock Tree	45
Figure 34: Clock Tree Example.....	46
Figure 35: <i>Back-End</i> → <i>Clock Declaration</i> Panel	46
Figure 36: <i>Back-End</i> → <i>Clock Handling</i> panel with default settings	49
Figure 37: <i>Timing Analysis Options</i> Frame.....	50
Figure 38: Automatic zCore Generation in the <i>Clustering</i> tab.....	52
Figure 39: <i>Back-End</i> → zCore Definition panel.....	53
Figure 40: zCore-level Performance Oriented Partitioning.....	56
Figure 41: <i>Clustering</i> panel showing the <i>Mapping Output File</i> frame	57
Figure 42: <i>Clustering</i> panel for <i>Automatic with Parameters</i> mode	58



Figure 43: <i>Clustering</i> panel for <i>Pre-existing Mapping</i> mode	59
Figure 44: <i>RTL-based Compilation Script</i> item in <i>Project</i> view	61
Figure 45: <i>Initial Value for Registers</i> in <i>Advanced</i> Panel	63
Figure 46: <i>Memory Sources</i> Panel.....	78
Figure 47: <i>Preferences</i> → <i>Memories</i> panel	79
Figure 48: LUTRAM- and BRAM-based Memory Port Functional Architecture	82
Figure 49: Memory Port Functional Architecture.....	83
Figure 50: Declaring the Hardware Configuration File (example with sample file)...	95
Figure 51: FPGA P&R File Policy	96
Figure 52: <i>Preferences</i> → <i>Job Scheduling</i> Panel	96
Figure 53: Choosing the compilation target	99
Figure 54: Tools Menu and Compiler Toolbar	100
Figure 55: Task Tree Pane during Compilation	101
Figure 56: External Facilities frame in <i>Preferences</i> → <i>Compiler</i> panel	103
Figure 57: Retargeting an ASIC Design for ZeBu	107
Figure 58: Declaring the source files when retargeting with GTECH.....	108
Figure 59: <i>Advanced</i> → <i>PARFF</i> frame	113
Figure 60: <i>Original</i> FPGA P&R task	114
Figure 61: Task tree showing relaunched and retried P&R tasks	114
Figure 62: Task tree showing FPGA P&R tasks	115
Figure 63: FPGA Parameters before any compilation.....	116
Figure 64: FPGA Parameters after first compilation	117
Figure 65: Filters for FPGA ISE Parameters.....	118

Tables

Table 1: Items of the <i>Project Tree</i>	16
Table 2: Criteria for Synthesizer Choice.....	23
Table 3: Memory Implementation Guidelines	77
Table 4: Signals Description for BRAM-based Memory Models.....	82
Table 5: Signals Description for zrm-based Memory Models.....	83
Table 6: Options for the <i>probe_signals</i> Command	90
Table 7: FPGA P&R Parameter Sets for PARFF	115



About This Manual

Overview

This manual first explains how to create the compilation project for an easy and efficient compilation with **zCui**, which provides the graphical interface for the ZeBu compilation flow, including synthesis, gate-level compilation and FPGA Place and Route.

In the compilation project, you have to define the best options for clock modeling, memory modeling and to choose the most efficient verification environment, targeting either the compilation time or the emulation runtime frequency. Specific recommendations are given for advanced modeling features which are not integrated in the **zCui** graphical interface.

Appropriate information is provided for analysis of the various compilation logs.



History

This table gives information about the content of each revision of this manual, with indication of specific applicable version:

Doc Revision	Product Version	Date	Evolution
c	V6_3_0	Dec 10	<p>Many modifications have been integrated in the present revision of this manual. The major modifications are listed here:</p> <p><u>New Sections:</u></p> <ul style="list-style-type: none">• Information for Additional Command Files (§5.5)• Parallel Automatic Recompilation of Failing FPGAs (PARFF) (§5.4.2)• Compiling to Optimize Runtime Performance (§5.3)• RTL-based Compilation Scripts (§3.7.2)• Flattening the Netlist for a translation library (§5.1.2)• Transactor Mapping (§3.4.10)• Configuring the allocation of clock resources for user clocks (§5.2.2)• Retry mechanism for NFS failures (§4.1.4)• Detection of Combinational Loops (§3.9.5)• Writing a Memory Source file (§3.8.6)• Selection in a Log for Copy-Paste (§2.3.2) <p><u>Other Updates:</u></p> <ul style="list-style-type: none">• Complete revision of the zCui snapshots.• Updated icons for compilation status (§2.3 and 4.3.1)• Added information about zFAST Script Mode (§3.2.3)• Complete revision of section “Distributing the design in the system” (§3.6).• Better explanations for force and force_dyn commands (§3.7.5 and 3.7.6)• Major updates for Memory Modeling (§3.8).• Modifications for compilation for relocation (§4.1.5).• Updated intro for ISE parameters modification from FPGA panel (§5.4.3).
b	6_2_0	Jan 10	<p><u>New sections:</u></p> <ul style="list-style-type: none">• ZeBu-Server Hardware Overview (§ 1.1).• Distributing the FPGA usage (§ 3.6).• Handling the design reset (§ 3.7.2).• Handling the Scan Chain (§ 3.7.4.1).• Dynamic Force (§ 3.7.5).• Options for storage of FPGA P&R intermediate files (§ 4.1.2).• Configuring the Compiler for Job Scheduling (§ 4.1.3)• Compiling a design for relocation (§ 4.1.4)• Monitoring successful compilation (§ 4.3.1.1), incl. FPGA statistics and timing analysis results. <p><u>Updated information:</u></p> <ul style="list-style-type: none">• Information about the compilation flow (§ 1.2).• Methodology description (§ 1.3).• Description of the Compilation View (§ 2.2.4).• Introduction about the compiler input files (§ 3.1.1).



Doc Revision	Product Version	Date	Evolution
			<ul style="list-style-type: none">• List of third-party synthesizers (§ 3.2.1).• Info about No. of FPGAs selector for third-party synthesizers (§ 3.2.4.2).• Complete review of the Verification Environment section (§ 3.4), in particular sub-sections re-ordering.• Explanations for zCores and examples (§ 3.6.1).• Output mapping file and clustering with pre-existing mapping (§ 3.6.2).• Explanations for clustering (§ 3.6.2).• Re-ordered the section about clustering settings (now § 3.6.2).• Information about the file to use for the design modeling commands (§ 3.7).• Information about tristates (§ 3.7.3).• Corrections for <code>force_dyn</code> command (§ 3.7.5).• Information in the Debug section (§ 3.9).• Information about the configuration file (§ 4.1.1).• Information for Job Scheduling commands (§ 4.1.3).• Re-ordered the sections about compilation target and compilation launching (§ 4.2.1 and 4.2.2).• Choice for HTML viewer in zCui (§ 4.4.1).
a	6_1_1	Oct 09	First Edition

Related Documentation

The ZeBu-Server documentation package is listed in Chapter 6 of this manual.

1 Introduction

1.1 ZeBu-Server Hardware Overview

ZeBu-Server is an extremely high-capacity system emulator with the easy setup and debugging associated with emulation. It is available in either a 2-slot or a 5-slot unit to handle designs from 10M to 200M ASIC gates and up to 5 simultaneous users. ZeBu-Server is also expandable in a multi-unit environment to accommodate up to 1 billion ASIC gates and up to 25 users.



Figure 1: ZeBu-Server 2-slot and 5-slot Units

In ZeBu-Server, the Design Under Test (DUT) is mapped onto one or several Xilinx Virtex-5 LX330 FPGAs and memory chips. The test environment is mapped onto dedicated FPGAs (interface FPGAs, IF) which implement the Reconfigurable Test Bench (RTB), an innovative interface technology made up of a combination of proprietary hardware, firmware and software layers. Through this technology, ZeBu-Server supports several software and hardware debugging modes that can undertake the most challenging verification problems presented by today's electronics products during the entire design cycle.

ZeBu-Server is a scalable system and different types of FPGA modules may populate the ZeBu-Server unit:

- 4C module:
 - 4 FPGAs with 2 Gbits of RLDRAM per FPGA to map the design
 - 1 FPGA with 4 GBytes of DDR2 memory to map the design
 - 1 FPGA to map the RTB
- 8C / ICE module:
 - 8 FPGAs with 1 Gbits of RLDRAM per FPGA to map the design
 - 1 FPGA with 2 GBytes of DDR2 memory to map the design
 - 1 FPGA to map the RTB
 - Direct ICE interface (1,200 data pins and dedicated clock pins) to connect a target system.
- 16C module:
 - 16 FPGAs to map the design
 - 1 FPGA with 4 GBytes of DDR2 memory to map the design
 - 1 FPGA to map the RTB

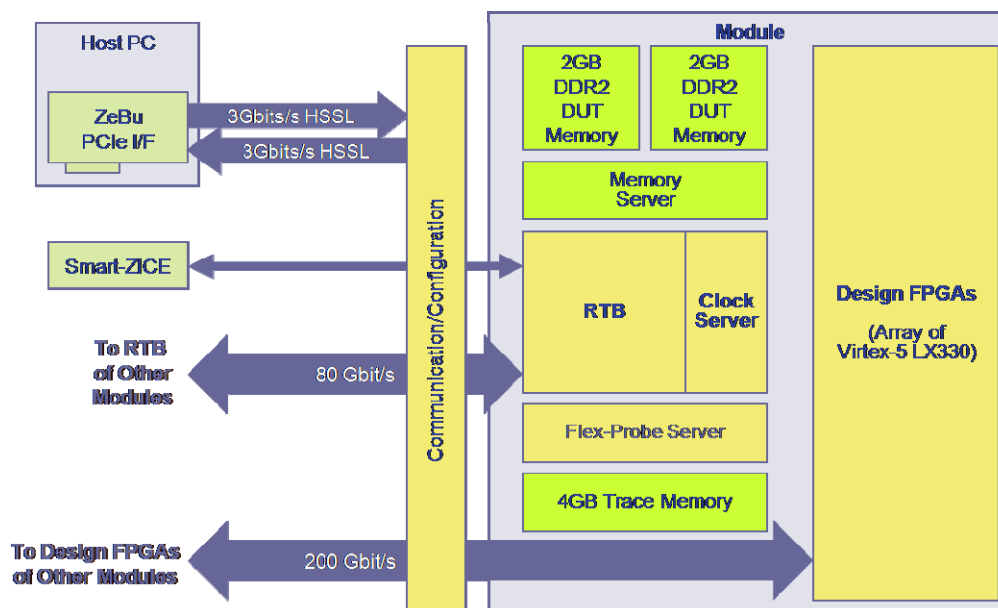


Figure 2: Architecture of an FPGA module

The ZeBu-Server unit is connected to the host PC through a PCI Express interconnection board. In multi-user environment, a different PC can be connected to each module of the unit.

The Smart Z-ICE interface (64 data pins and 4 clock pins for a 2-slot unit; 80 data pins and 5 clock pins on a 5-slot unit) provides the support of standard software debuggers such as JTAG cables. This interface and its usage are fully described in the [ZeBu-Server Smart Z-ICE Manual](#).

The Direct ICE interface is available to connect the DUT to a target system or a hardware IP core via 1,200 data pins and dedicated clock pins. This interface and its usage are fully described in the [ZeBu-Server Direct ICE Manual](#).

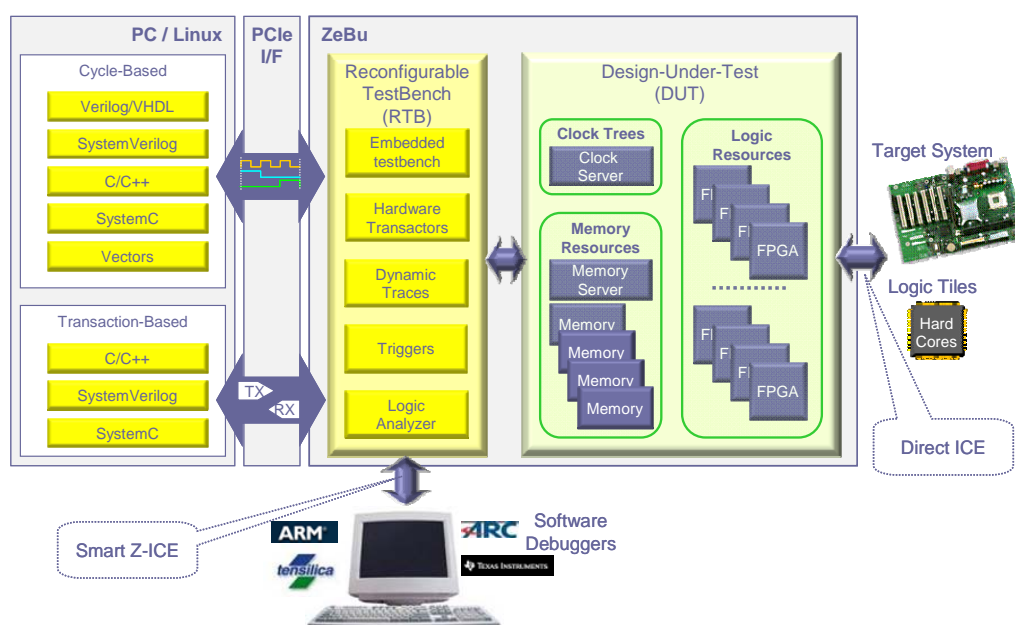


Figure 3: ZeBu-Server Runtime Architecture

1.2 Compilation Flow

The ZeBu compiler starts with the original ASIC Register-Transfer Level (RTL) code of the design and automates all of the necessary steps to synthesize and map the design for a given ZeBu hardware configuration.

The ZeBu compilation is a 4-stage process:

1. Front-end: Synthesizes RTL source files to generate gate-level netlists.
2. System-level back-end compilation: splits extra-large netlists into medium-sized netlists; processes system-level place and route and static timing analysis; generates the appropriate databases for runtime environment.
3. zCore-level back-end compilation: maps medium-sized netlists onto ZeBu technology, including clock mapping.
4. FPGA Place and Route: creates the binary files for the Virtex FPGAs.

zCui, the ZeBu compilation interface, provides a convenient graphical interface to create a ZeBu compilation project for your design and to control the compilation process, from RTL source files of the design to runtime binary files.

The following figure shows the main steps of the ZeBu compilation in **zCui** with the user input files.

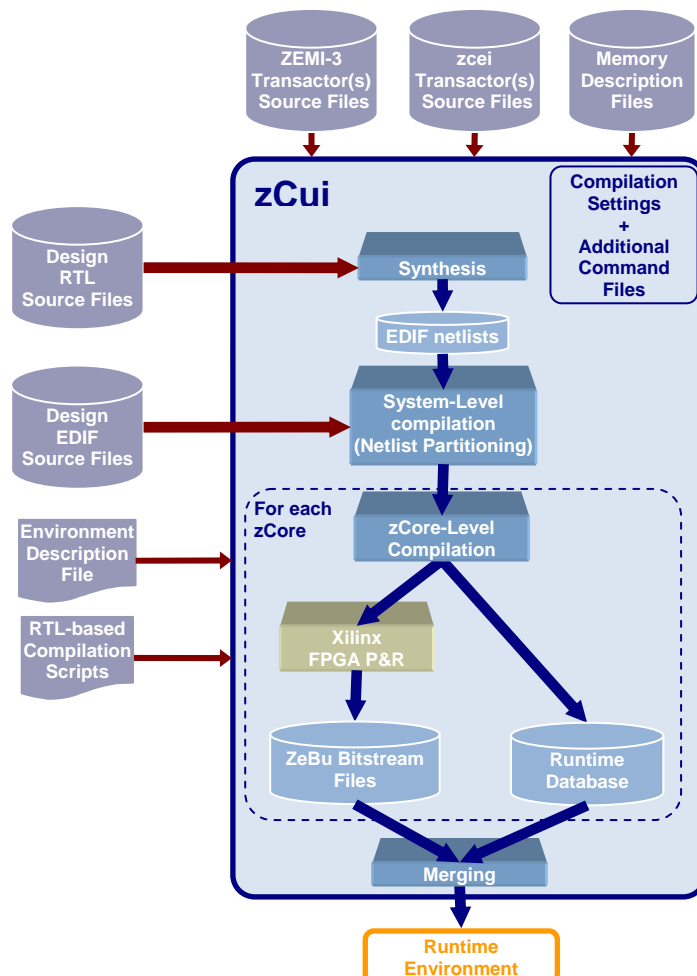


Figure 4: Compilation Flow



1.3 Compilation Methodology

The choices when creating the compilation project are important because a correct compilation project makes design verification easier. To create an optimized compilation project which balances compilation efficiency and emulation performance, some choices have to be made as early as possible in the verification process:

- Choice of the most appropriate verification environment (co-simulation, transactional, hardware speed rate adapters, etc.), considering the constraints of each environment.
- Synthesis strategy for the RTL source files of the design.
- Back-end compilation strategy, in particular for big designs.
- Clock modeling including connection to ZeBu primary clocks and strategy when the design includes PLLs or DCMs.
- ZeBu specific handling.
- Memory modeling, including appropriate mapping of DDR memory models.
- Decision about the debugging strategy.

The graphical interface of the ZeBu compiler, **zCui**, integrates the appropriate items to create and modify the compilation project; advanced modeling features can be controlled through additional command files.

With **zCui**, the compilation can be launched either from the graphical interface or in batch mode when the compilation project already exists. **zCui** automatically configures the ZeBu tools and the compilation results can be analyzed in the graphical interface, with filtering and search capabilities.

2 Quick Guide for zCui Interface

2.1 zCui Opening View

The **zCui** interface is a paned window which opens by default in the following view:

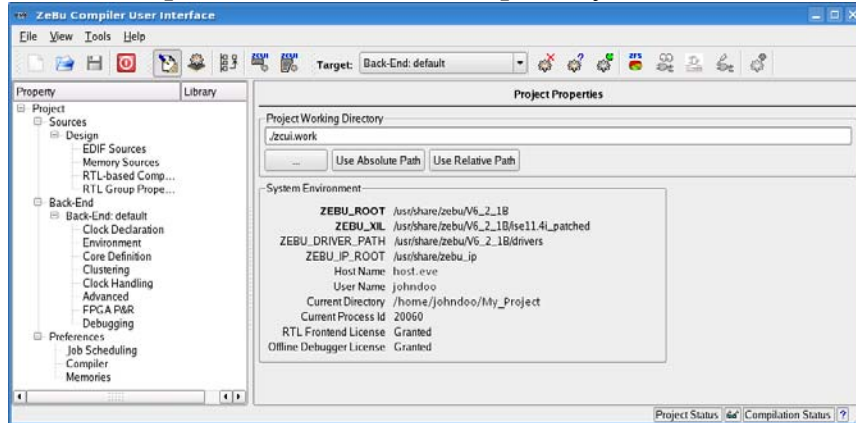


Figure 5: Opening display (Project View)

The **System Environment** frame displays information about your configuration. The values displayed in Figure 5 are for a standard environment.

zCui graphical interface has two different views (or workspaces):

- The **Project** view which is the opening display shown in Figure 5 in which the parameters for the compilation are set. The compilation settings are stored in a **zCui** project file (.zpf file).
- The **Compilation** view is displayed when launching the compilation and allows monitoring the compilation process and analyzing the logs:

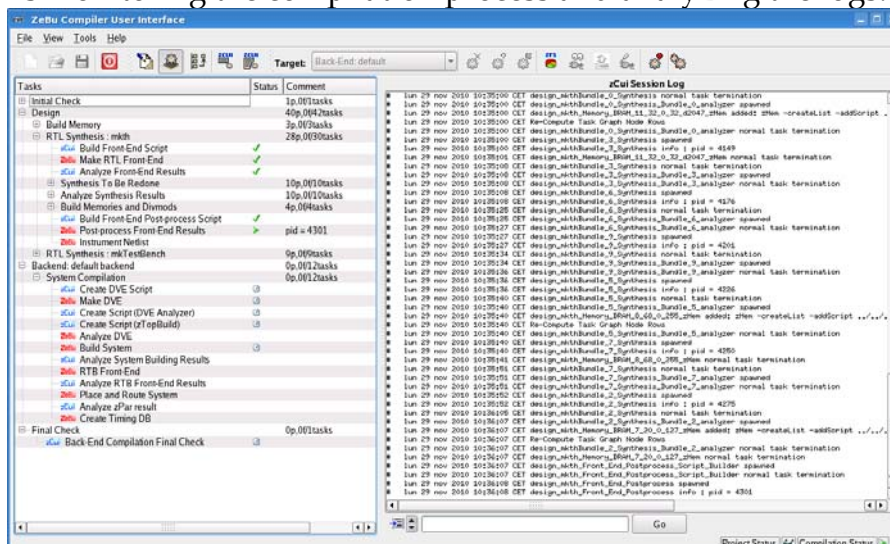


Figure 6: Compilation View

In both **Project** and **Compilation** views, the **zCui** GUI is a paned window with the usual horizontal bars at the top (title bar, menu bar and toolbar). The rest of the window is divided vertically into two expandable left and right panes which are different in the two **zCui** workspaces.

2.2 Description of the zCui Views

The screen terminology of this chapter will be used throughout this manual.

2.2.1 Project View

The **Project** view shows the **Project Tree** pane and the **Properties** pane. These panes can be resized by moving right/left the vertical separator, and they can even span the full width of the GUI. When necessary, horizontal and vertical scroll bars are added.

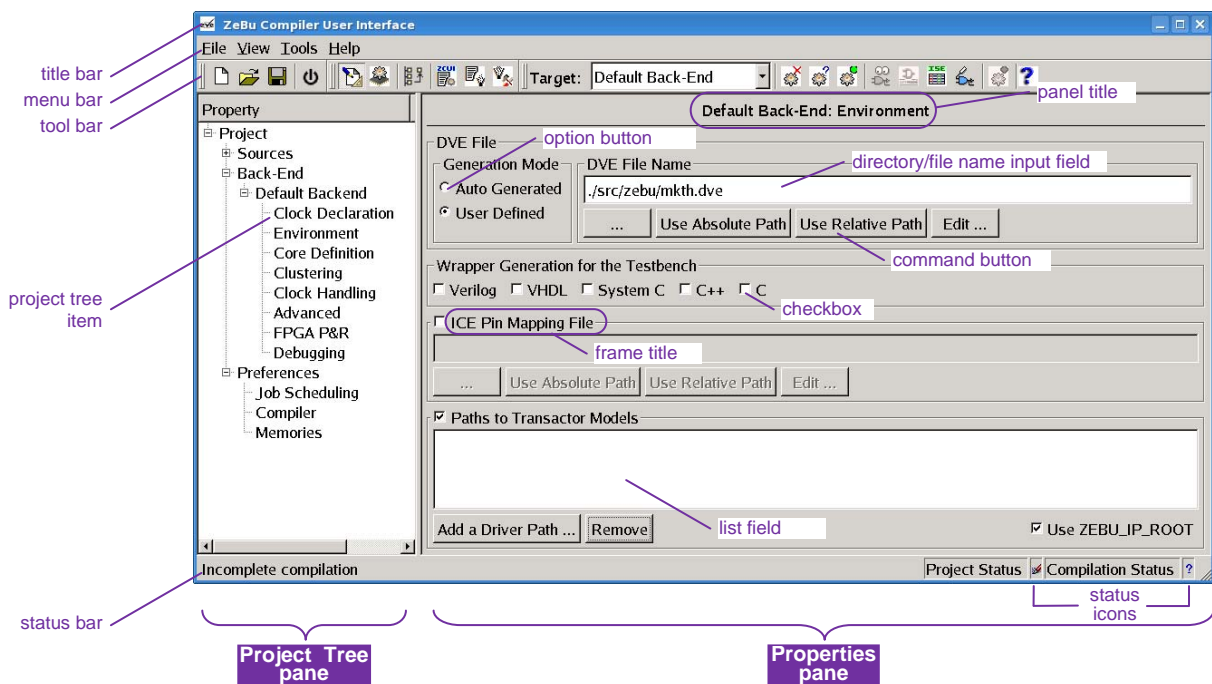


Figure 7: Details of the Project View with a Properties Pane

The **Properties** pane is replaced by the **Inspector** pane when a source file item (RTL or EDIF source file, memory script, RTL-based compilation script) is clicked in the **Project Tree** pane.

The **Project Tree** pane is a tree-like view of all the items which belong to the project workspace. Clicking on any item in the **Project Tree** pane brings up a specific properties panel in the **Properties** pane.

Table 1: Items of the Project Tree

Project Tree item		Description
Project		Working directory and environment settings
Design	(root)	Declaration of the top-level of the design
	EDIF Sources	Declaration of the EDIF source files of the design
	Memory Sources	Declaration of the Tcl Memory Description Files and compilation settings for memories



Project Tree item		Description
Design	RTL-based Compilation Scripts	Declaration of Tcl scripts which include commands supporting hierarchical paths to nets and instances with the name used in the RTL source files. The supported commands are different according to the synthesizer, as described in Section 3.7.2.
	RTL Group Properties: Group_Name	Declaration of the RTL source files of the design and parameters for synthesis (several tabs)
ZEMI-3 Transactor	(root)	ZEMI-3 dedicated parameters
	EDIF Sources	Declaration of the EDIF source files of the ZEMI-3 transactor
	Memory Sources	Declaration of the Tcl Memory Description Files and compilation settings for memories
	RTL Group Properties: Group_Name	Declaration of the RTL source files of the transactor and parameters for synthesis (several tabs)
Transactor	(root)	Top name of the transactor
	EDIF Sources	Declaration of the EDIF source files of the design
	Memory Sources	Declaration of the Tcl Memory Description Files and compilation settings for memories
	RTL Group Properties: Group_Name	Declaration of the RTL source files of the transactor and parameters for synthesis (several tabs)
Back-End	Back-End_Name (BEx)	Declaration of the target hardware configuration file and settings for static timing analysis
	Clock Declaration	List of design clocks with their types
	Environment	Declaration of the verification environment: several tabs for DVE file, co-simulation wrappers, Direct ICE files and information for external transactors
	zCore Definition	Declaration of the netlist edition file and zCore definition file
	Clustering	Parameters for clustering
	Clock Handling	Parameters for clock modeling
	Advanced	Specific parameters for the compiler, in particular additional command files
	FPGA P&R	Parameters for Xilinx ISE FPGA P&R tools
	Debugging	Parameters for debugging capabilities
Preferences	Job Scheduling	Parameters for the compilation processing environment
	Compiler	
	Memories	

When source files have been added, clicking on a source file in the **Project Tree** pane displays the content of the source file in the **Inspector** pane:

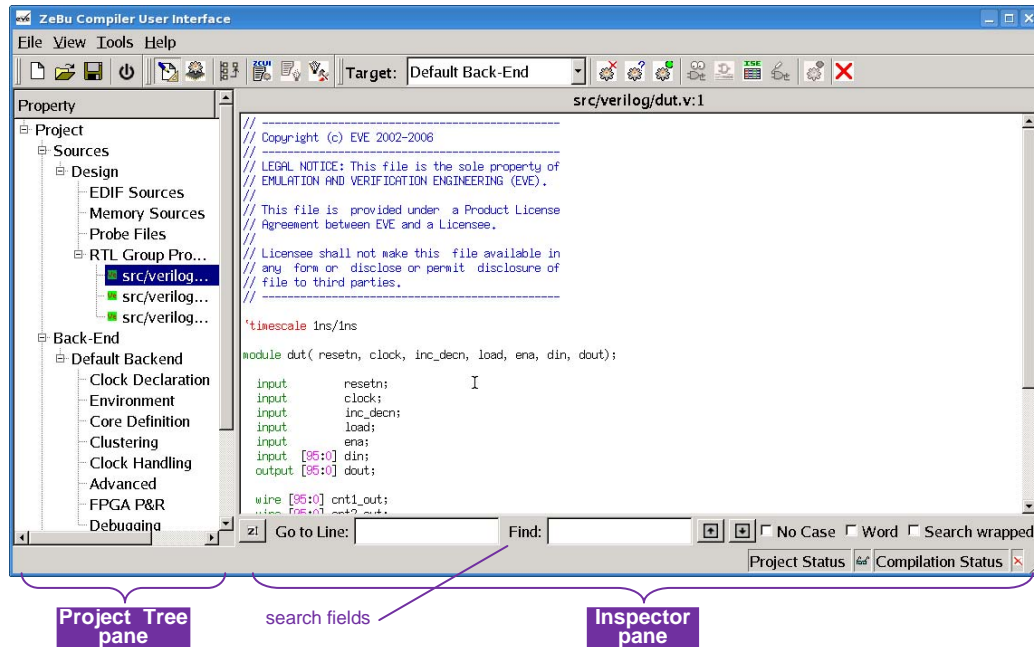


Figure 8: Details of the Project View for a Source File (Inspector pane)

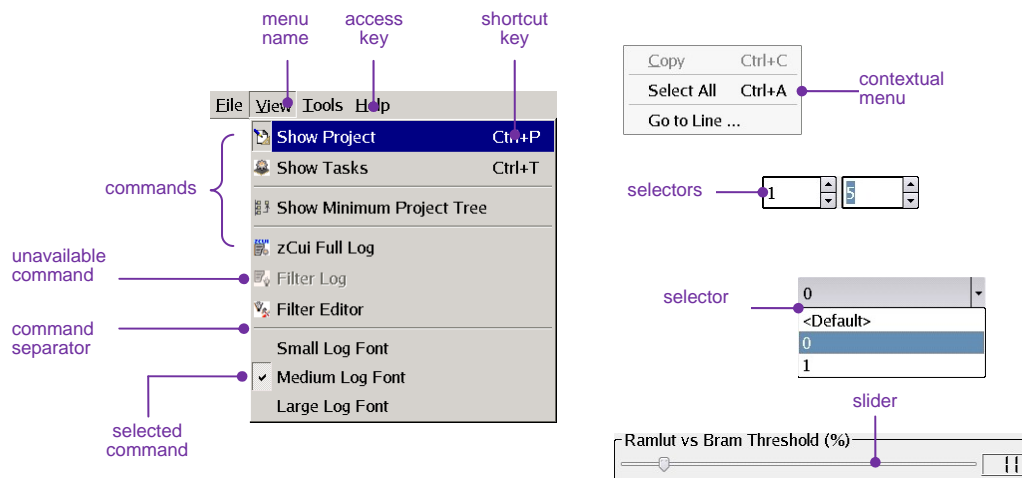


Figure 9: Additional Elements in zCui GUI

2.2.2 Menu bar

The **zCui** menu bar contains the following items:

- **File:** **zCui** project file management usual operations (Create, Open, Save, Save As, Import, Export, Archive, Exit).
- **View:** Toggles **Project** to **Task** views and provides settings for display.
- **Tools:** Direct access to Compilation tools.
- **Help:** the **About...** menu provides information on the current version of **zCui**.

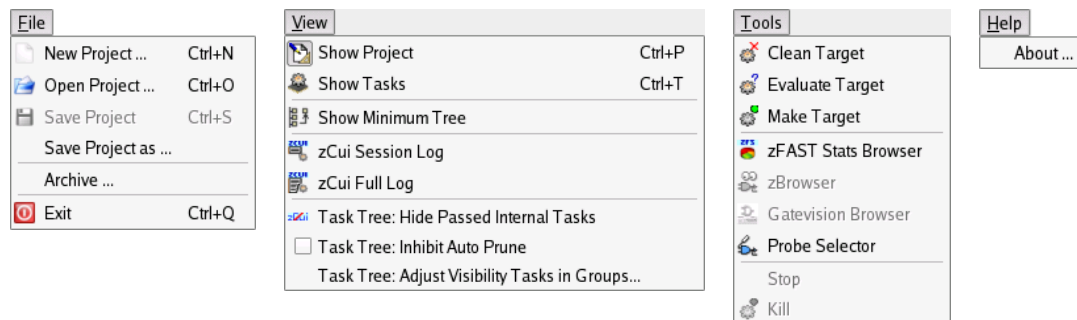


Figure 10: zCui Menus

2.2.3 Toolbars

The **zCui** toolbar is a set of 3 separate toolbars:

- **File Operations** toolbar
- **View** toolbar
- **Compiler** toolbar

You can choose to hide/display any toolbar in the toolbar zone. To do that, bring up the toolbar contextual menu by right-clicking in the toolbar zone:

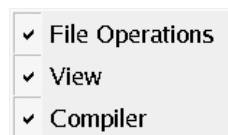


Figure 11: Toolbar Contextual Menu

You can also drag any toolbar from its handle and drop it anywhere inside or outside the GUI, or even change the order of toolbars in the toolbar zone.

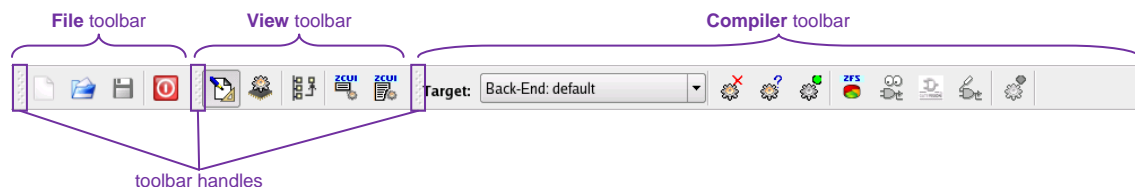




Figure 12: zCui Toolbars

The **File** and **View** toolbars provide the basic operations from the corresponding **File** and **View** menus.

2.3 Compilation View

When the compilation is launched, when evaluating the project or when generating the DVE file, **zCui** automatically switches to the **Compilation** view. You can toggle between **Project** and **Compilation** views by clicking the relevant icon in the toolbar:  for **Project** view and  for **Compilation** view. Note that the **Compilation** view is sometimes named **Task** view.

Like in **Project** view, two panes are displayed in **Compilation** view (**Task Tree** and **Log** panes). These panes can be resized by moving right/left the vertical separator, and they can even span the full width of the GUI. When necessary, horizontal and vertical scroll bars are added.

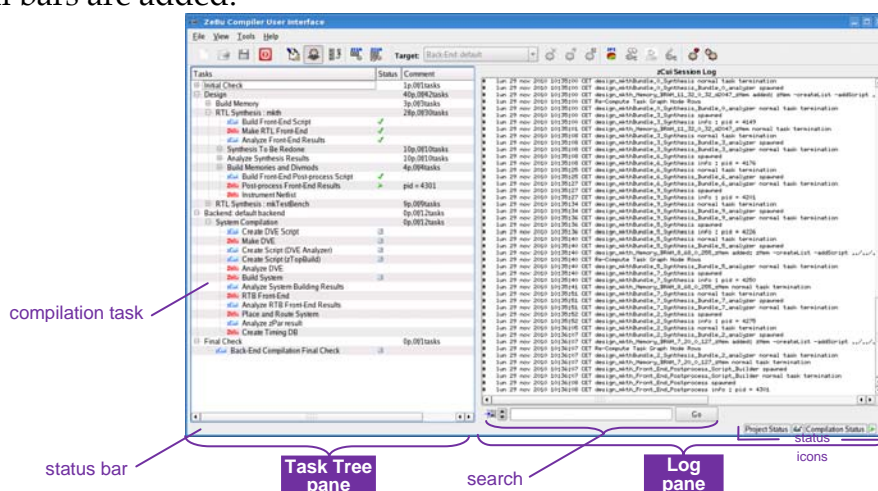









Figure 13: Compilation View

For each compilation task, the **Task Tree** pane shows the current status:

-  Pending task (not yet launched by **zCui**).
-  Task in progress.
-  Completed task; successfully finished.
-  Completed task; failed.
-  Task with missing inputs (only for the **Evaluate** operation).
-  Group in progress but 1 task (or more) of the group failed.
-  Cancelled pending task (in case of PARFF, as described in Section 5.4.2).

Note that the same icons are used for **Compilation Status** at the bottom-right corner.

When right-clicking on a completed compilation task in the **Task Tree** pane, it is possible to display specific logs and reports of the compilation process (for example for the **Build System**), access the logs/reports in a separate editor (**Show Log in a Separate Window**) or relaunch a task which failed (**Redo**).

Note that the **Redo** option is not available for some of the compilation tasks, in particular FPGA Place & Route tasks.

The **Log** pane includes some search and filtering features for easier investigation in the compilation results, as described in Section 2.3.1.

Additional information about the content of the **Compilation** view is available in Section 4.2.2.

2.3.1 Search Tools for Source files and Logs

zCui provides search capability in the source files and in the logs with different criteria. The toolbar at the bottom of the source/log pane shows different icons for these criteria:

- Go to Line (🔍): this is the default mode.
- Search for text (🔍)
- Filtering (🔍)
- Error Navigation (🔍)

A specific toolbar (🔍) is available to modify some viewing properties such as the font size and automatic scrolling for the current source/log file.

To change the active search toolbar, use the up-down arrows placed next to the search criteria icon:

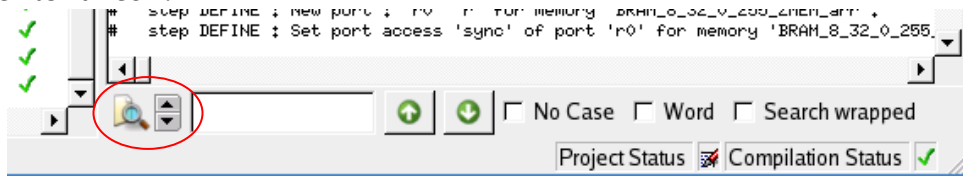


Figure 14: Search Toolbar

Note that the source/log files can also be edited with a separate editor by selecting **Show in a separate window** in the contextual menu of the corresponding task.

The editor which displays the source/log files can be set by user in the **Preferences** → **Compiler** table (**External facilities** frame) in the **Project** view.

2.3.2 Selection in a Log for Copy-Paste

When selecting some text for copy-paste purposes in a log displayed in **zCui**, the complete line is automatically selected and it is not possible to copy only a portion of this line.

Three copy-paste methods are supported by **zCui** (some limitations may exist according to your desktop environment and/or workstation setup, in particular when using a VNC environment):

- Mouse method: after selection, use the mouse wheel in the target application to paste.
- Paste buffer: after selection, CTRL+C and CTRL+V to paste if the target application supports it.
- Drag-and-Drop: after selection, drag the text with the mouse and drop it in the target application (note that it is not a Move but a Copy by default since the selected text remains available in **zCui**). The text is copied as plain text.

Note that there is no scrolling capability when selecting text: only the text currently displayed in the log pane can be selected; to copy a longer portion of the log, it is necessary to proceed with several copy-paste operations.



3 Preparing the Compilation Project

3.1 Introduction

3.1.1 Inputs of the ZeBu Compiler

In the ZeBu Compiler, the source files which are visible in Figure 4 are declared for the design itself and for the ZEMI-3 and zcei transactors of the environment:

- RTL source files: Verilog, SystemVerilog or VHDL files.
- Gate-level EDIF netlists, synthesized out of the ZeBu compiler.
- When some design memories need to be generated by the ZeBu Memory Generator, the appropriate Tcl files are declared in the **Memory Sources** item. Note that when synthesizing with **zFAST**, such memory source files are not necessary because the memories instantiated in the RTL source files are automatically processed by the ZeBu compiler.
- Declaration of probes for runtime debugging and specific compilation commands with the RTL paths to the signals in the **RTL-based Compilation Scripts**, as described in Section 3.7.2.

The ZeBu compiler requires a file which describes the connection of the design with its verification environment. This mandatory file is known as the “DVE file”, which stands for “Design Verification Environment”. It includes the appropriate declarations for the design clocks and for the drivers/monitors that will be used to test the design. Some detailed information about the DVE file is available in Section 3.4.1 of the present manual.

3.1.2 Organizing the Input Files

In order to keep any pre-existing environment for later reference, in particular a pre-existing simulation environment, it is strongly recommended to copy the complete file tree of the design in a ZeBu dedicated directory, so that the ZeBu compilation does not use the same physical files as the pre-existing environment.

It is also recommended to store the ZeBu dedicated files (DVE file, memory description files, RTL-based compilation scripts, additional command files, **zCui** project file) in a separate directory.

The following file tree shows a possible organization for the input files:

```
<My_Project>
├── src
│   ├── rtl
│   │   └── (...)
│   ├── edif
│   │   └── (...)
│   └── zebu
│       ├── my_project.zpf
│       ├── my_project.dve
│       ├── probes
│       │   └── (...)
│       └── memories
│           └── (...)
```

With this file tree, **zCui** should be launched from `<My_Project>/` directory so that the compilation working directory is created at the same level as the `src/` directory. Detailed information about how to launch **zCui** is available in Section 4.1.



3.2 RTL Source Files

3.2.1 Choosing the synthesizer

The ZeBu compiler offers the choice between the EVE proprietary synthesizer, known as ZeBu Fast Synthesis (**zFAST**), and usage of a third-party FPGA synthesizer with an integrated ZeBu RTL Front-End (**zRtlFrontEnd**).

The following third-party FPGA synthesizers are supported:

- Synplify Pro, Synplify Premier and Synplify Premier DP from Synopsys (<http://www.synopsys.com>).
- XST from Xilinx (<http://www.xilinx.com>), except for ZEMI-3 transactors.

The following table lists the most relevant features to compare **zFAST** and third-party synthesizers integrated with **zRtlFrontEnd**:

Table 2: Criteria for Synthesizer Choice

Feature	zFAST	Third-Party Synthesizer with zRtlFrontEnd
Faster synthesis	✓	
Better logic area		✓
Verilog	✓	✓
VHDL	✓	✓
Automatic VHDL sorting	✓	✓
SystemVerilog	✓	✓
Mixed language	✓	✓
ZEMI-3 transactors	✓	✓
Block-based synthesis	✓	✓
Top-down synthesis	✓	✓
Parallel Synthesis	✓	✓
SystemVerilog Assertions (SVA)	✓	
Hierarchical references	✓	✓
User Defined Primitives (UDP)	✓	✓
Combinational Simulated Signals (CSA)	✓	
Tracers based in Import DPI (zDPI feature)	✓	
Memory automatic inference	✓	
Memory initialization with \$readmemh	✓	

zFAST offers two different operating modes which are fully described in the **zFAST Synthesizer Manual** (Rev A):

- With **zFAST** standard mode, the declaration of source files and most of the synthesis options are set through the **zCui** graphical interface. Only advanced attributes are declared with an additional file.
- The **zFAST** script mode is based on input command files very similar to the simulation scripts. Using the **zFAST** script mode provides more flexible options for the analysis and elaboration steps before synthesis.

For both modes, chunks and bundles are taken into account to launch the synthesis tasks.

Third-party synthesizers can also be used to generate EDIF netlists out of the ZeBu flow: the resulting EDIF files are declared as inputs for the ZeBu compiler (see 3.3).

3.2.2 Supported RTL Source Files

The ZeBu compiler supports both Verilog (Verilog IEEE 1364-2001/1995) and VHDL (VHDL IEEE 1076-1997/1987) RTL input files, which can be associated in mixed Verilog/VHDL designs.

SystemVerilog 3.1/3.1a (IEEE 1800-2005) is also supported, but some limitations exist according to the synthesizer.

When synthesizing with **zFAST**, the following SystemVerilog-based constructs are supported: SystemVerilog Assertions (as described in Section 3.9.3), DPI calls in the source files of the design (zDPI feature, described in a separate Application Note).

With any synthesizer supported by ZeBu, the language of the RTL input files can be detected automatically if it is not explicitly declared and you can declare RTL libraries.

3.2.3 Declaring the RTL source files

The RTL source files are gathered in one or several RTL Groups. By default, **zCui** creates one RTL group for the design (**Default_RTL_Group**).

It is recommended to have only one RTL group for the design, since most of the ZeBu debugging features are supported for one group only, in particular SystemVerilog Assertions, RTL-based compilation scripts and CSA feature (Combinational Signals Accessibility). Note that hierarchical references in the RTL sources are supported only inside an RTL group (hierarchical references between different RTL groups cannot be synthesized).

Additional RTL groups can be of interest when integrating RTL sub-projects or IPs, when investigating synthesis issues or when a sub-module needs some specific synthesis features for performance purposes in particular.

Each RTL group has its own synthesis options and synthesis output files are automatically merged by the ZeBu compiler after synthesis for back-end compilation.

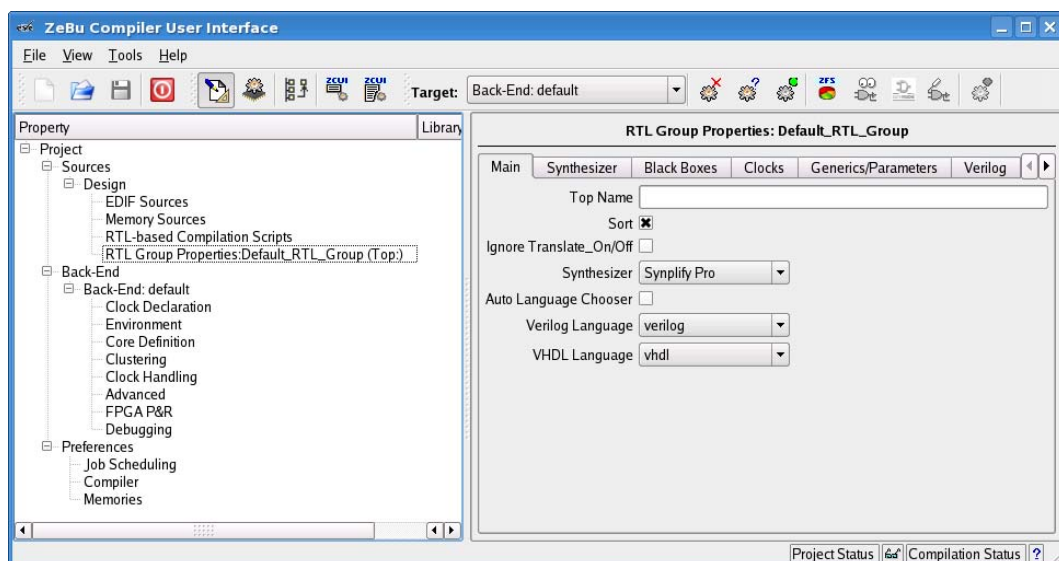


Figure 15: RTL Group Properties panel for the Default RTL Group

Except for **zFAST** script mode, the RTL source files can be added in each RTL group through contextual menus which are available by right-clicking on the RTL group name in the **Project Tree** or on an existing RTL source file of the RTL group:

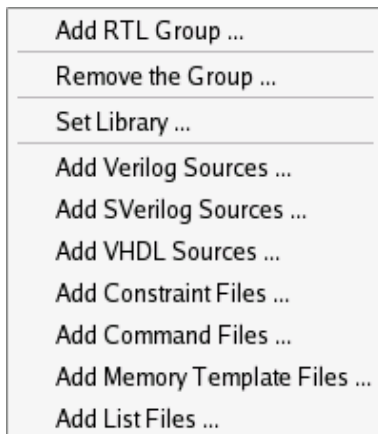


Figure 16:

Contextual menu for an RTL group

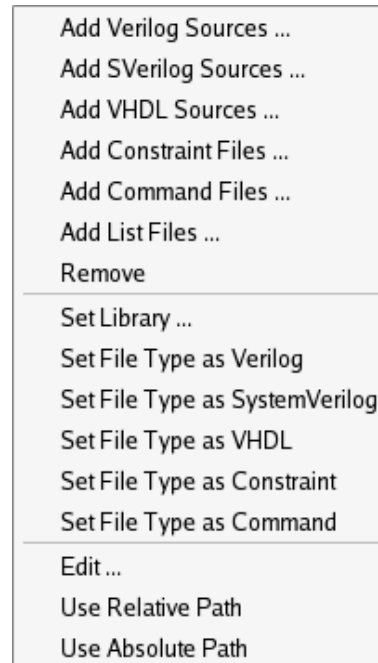


Figure 17: Contextual menu for an RTL source file

When selecting **Add <Language> Sources ...** (where **<Language>** can be **Verilog**, **SVerilog** or **VHDL**), a file browser window opens in which you can select the source files.

The default file extensions in the browser are different for each language:

Menu item	Default Extensions for the browser
Add Verilog Sourcesv, .vm, .vl, .vc
Add SVerilog Sourcesv, .vm, .vl, .vc, .sv, .slg
Add VHDL Sourcesvhd, .vhdl, .VHD, .VHDL

It is possible to declare the RTL source files from a separate file which lists the RTL source files: choose **Add List Files...** and all the source files are added as if they had been added individually. This feature may be of particular interest for a sorted list of VHDL files.

Note that **Add Constraint Files...** and **Add Command Files...** items are relevant for **zFAST** script mode and third-party synthesizers (not for **zFAST** standard mode).

An icon in front of the file name in the **Project** tree shows the type of each source file:

-  for Verilog
-  for SystemVerilog
-  for VHDL

In the contextual menu for an RTL source file (see Figure 17), some additional items are available:

- If an inappropriate file has been added, you can remove it with the **Remove** item.
- If the type of a source file (Verilog, SystemVerilog or VHDL) is not correct, you can change it with the **Set File Type as <file type>** item which can be seen in Figure 17. The file type icon in front of the file name is automatically changed.
- When the content of an RTL source file has to be modified, you can edit the file with vi editor from **zCui** (**Edit...** item in the contextual menu, Figure 17). The editor can be changed in the **Preferences** → **Compiler** panel, as described in section 4.4.1.

When there are several RTL groups, you can drag and drop a source file from one group to another.

When VHDL source files have to be sorted manually, you can drag and drop a source file in the list to set the order correctly.

The following figure shows the **Project** view when RTL files have been added:

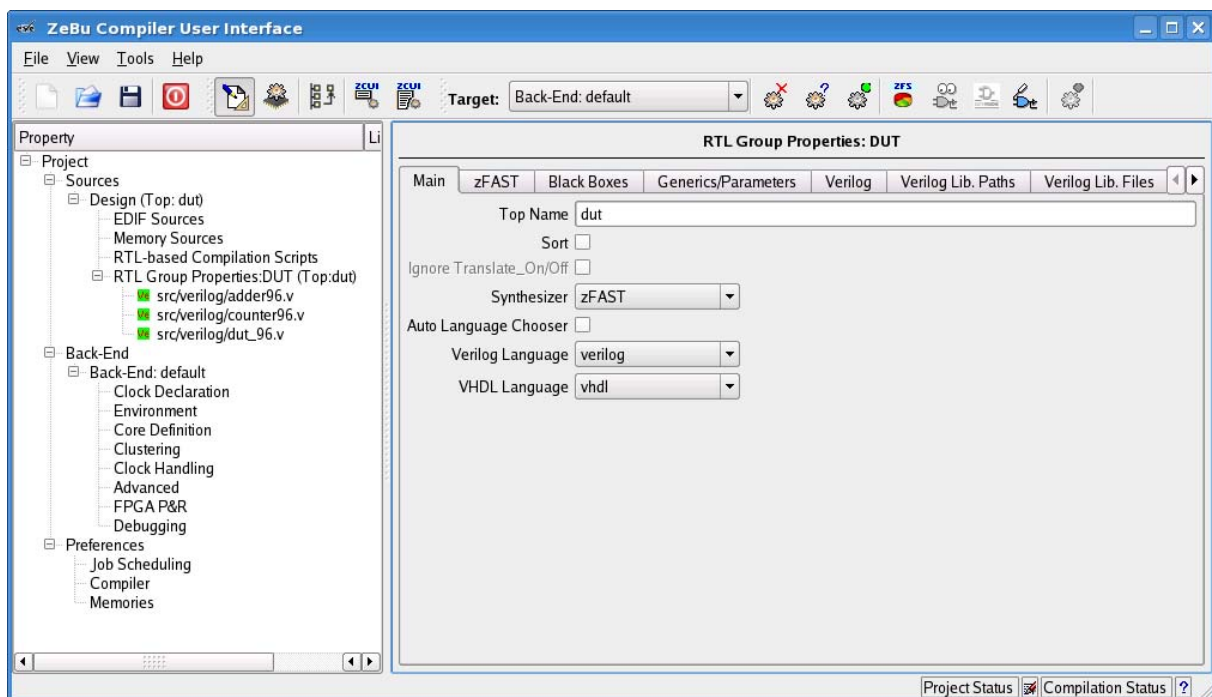


Figure 18: RTL Group Properties panel and RTL source files added

3.2.4 Defining the Synthesis Options

The choice for the synthesis options is different with the emulation performance and the debugging level that is targeted.

The following sections present the declaration of synthesis options for **zFAST** standard mode and for third-party synthesizers. The **zFAST** script mode is described in the [zFAST Synthesizer Manual](#).

3.2.4.1 Settings for **zFAST** standard mode

This section gives the basic settings to synthesize a design with the **zFAST** standard mode. Additional information is available in the [zFAST Synthesizer Manual](#), in particular for the script mode and for advanced features

When **zFAST** is selected to synthesize an RTL group, the **RTL Group Properties** panel for this group looks like the following example:

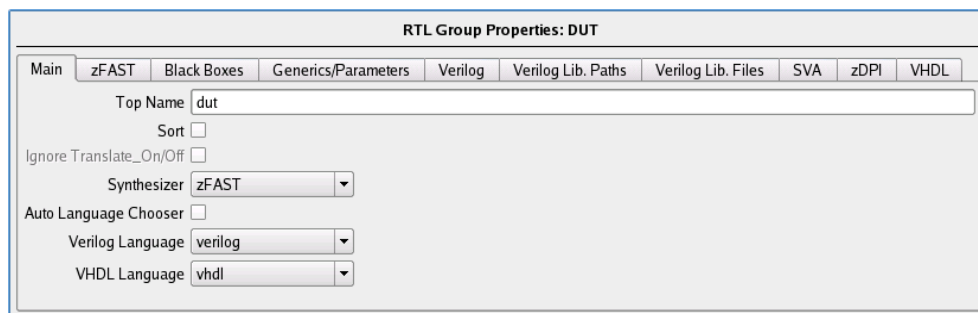


Figure 19: RTL Group Properties → Main tab for zFAST

When **zFAST** is the selected synthesizer, the following settings are available:

- When there is only one RTL group for the design, the **Top Name** field can be left blank: **zFAST** uses the **Top Name** declared in the **Design** panel. When there are several RTL groups, the **Top Name** field is mandatory in the **RTL Group Properties** panel for each group.
- The **Sort** checkbox is selected by default to provide automatic sorting of VHDL files so that the order in which the VHDL files are added in **zCui** does not impact the synthesis. If this checkbox is cleared, the VHDL files have to be sorted correctly by user before synthesizing (drag and drop the files displayed in the list of the RTL group in the **Project** tree pane to get the appropriate order).
- When some source files of different types (Verilog and SystemVerilog or Verilog and VHDL) have been declared without using the appropriate file type, the **Auto Language Chooser** checkbox can be selected so that the synthesizer automatically analyzes the files for the appropriate syntax. The file type selection is done on the file extension. This option is cleared by default.

- The **Verilog Language** and **VHDL Language** selectors are recommended when the RTL source files are not compatible with the default settings:
 - Verilog-2001 is the default for Verilog files. Other possible choices are Verilog-95 and SystemVerilog if source files are not compatible with Verilog-2001.
 - VHDL-93 is the default for VHDL files. Other possible choice is VHDL-87 if source files are not compatible with VHDL-93.

Additional settings can be modified in the other tabs of the **RTL Group Properties** panel:

- The **zFAST** tab includes specific options for **zFAST**, in particular the choice between block-based and top-down synthesis and the settings for optimization and debugging capability. It also provides the capability to declare an **Additional zFAST Attribute File** for options which are not available in the **zCui** graphical interface.
- When the design includes SystemVerilog Assertions and/or synthesizable DPI calls, the corresponding settings are located in the **SVA** and **zDPI** tabs, as described in the dedicated Application Notes.
- The other tabs (**Generics/Parameters**, **Verilog**, **Verilog Lib. Paths**, **Verilog Lib. Files**, **VHDL**) match the usual settings of most simulators.

By default, the **zFAST** tab looks like the following figure:

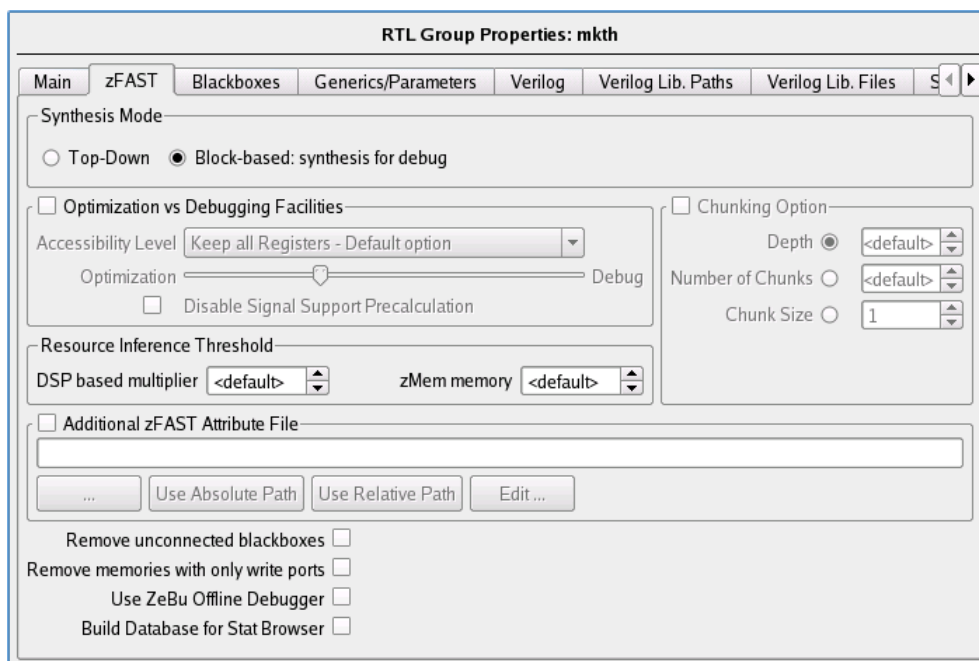


Figure 20: RTL Group Properties → zFAST tab

Block-based is the default value for the **Synthesis Mode** frame because it is the most appropriate mode when debugging the design.

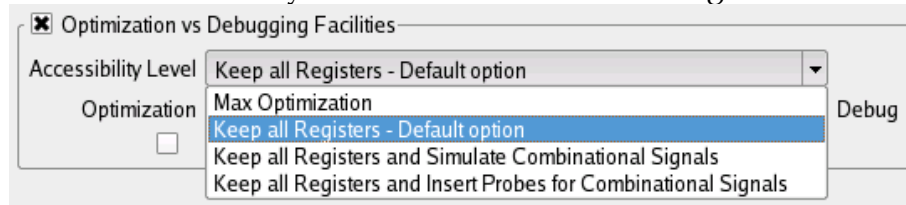
Selecting **Top-Down** mode is recommended when targeting runtime performance or area optimization.

When **Top-Down** mode is selected, selecting the **Chunking Option** checkbox creates separate synthesis processes by gathering the RTL modules of the group. The modules are gathered with one of the following criteria:

- Number of hierarchical levels to synthesize in the same process (**Depth**, which is the default setting).
- Number of separate synthesis processes (**Number of Chunks**).
- Total size of the modules that will be gathered (**Chunk Size**), based on the number of lines in the source files. The ratio for the value in the selector is 1 unit per 100 lines of codes.

The chunks are synthesized in parallel if the **Number of Jobs** value is higher than 1 for **Synthesizer** in the **Preferences** → **Job Scheduling** → **Process Scheduling** frame (see Section 4.1.3) and if several synthesizer licenses are available.

The **Optimization vs Debugging Facilities** frame offers the capability to balance the use of logic resources with easy runtime access to internal signals of the design.



The possible values in the **Accessibility Level** selector are the following ones:

- **Max Optimization:** some registers may not be accessible at runtime.
- **Keep all Registers - Default option:** all registers are accessible. Combinational signals are not accessible, except if dynamic probes are inserted manually in a RTL-based compilation script.
- **Keep All Registers and Simulate Combinational Signals:** all signals are accessible; the value of combinational signals will be simulated from register values.
- **Keep All Registers and Insert Probes for Combinational Signals:** all signals are accessible; probes will be added for combinational signals, which will increase the effective size of the design.

The below slider (**Optimization** ↔ **Debug**) which can be seen in Figure 20 can also be used to modify the **Accessibility Level** value.

The selectors for **DSP based multiplier** and **zMem memory** in the **Resource Inference Threshold** frame should not be modified and keep **<default>** value in the selectors. They can be modified if the compilation failed due to FPGA resource issues.

An **Additional zFAST Attribute File** can be declared in the dedicated frame. The supported attributes and the syntax are described in the [zFAST Synthesizer Manual](#). Once declared, you can edit the file with vi editor from **zCui** (click **Edit...**). The default editor can be changed in the **Project Compiler Preferences** panel, as described in Section 4.4.1.

Additional checkboxes are also available in this panel to **Remove unconnected blackboxes** and to **Remove memories with only write ports**. In most cases, such elements in the RTL design are reserved for testing; they are not relevant for emulation and should not be processed by the ZeBu compiler. By default, these options are not selected.

The **Use ZeBu Offline Debugger** checkbox activates the appropriate compilation settings to support this feature at runtime.

The **Build Database for Stat Browser** checkbox must be selected if you intend to use the **zFAST** Stat Browser (described in the *[zFAST Synthesizer Manual](#)*). If the design has been synthesized with this checkbox cleared and you want to use the **zFAST** Stat Browser, you have to select this checkbox and relaunch the compilation with the **Design** target: only the corresponding step is launched.

3.2.4.2 Settings for Third-Party Synthesizers

This section gives the basic settings to synthesize a design with a third-party synthesizer using **zRtlFrontEnd**.

When a third-party synthesizer is selected for an RTL group, the **RTL Group Properties** panel for this group looks like the following example:

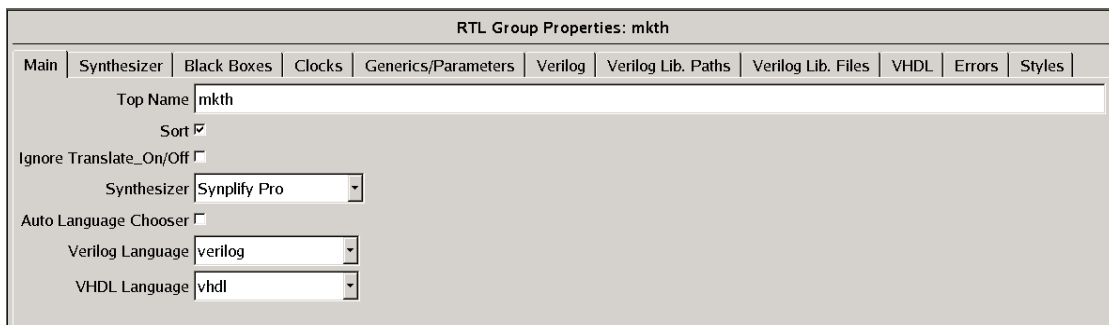


Figure 21: RTL Group Properties → Main tab for third-party synthesizers

When a third-party synthesizer is selected, the following settings are available:

- The **Top Name** field is optional. It can be left blank in most cases, in particular if there is only one RTL group and the **Top Name** has been declared in the **Design** panel.
In most cases, the RTL Front-End can automatically select the top level of a group from the content of the RTL source files.
The **Top Name** field should be filled in when the RTL source files include several configurations of the same design, in particular for bus modeling, or when testing with a synthesizable testbench.
- The **Sort** checkbox is selected by default to provide automatic sorting of VHDL files so that the order in which the VHDL files are added in **zCui** does not impact the synthesis.
If this checkbox is cleared, the VHDL files have to be sorted correctly by user before synthesizing (drag and drop the files to get the appropriate order).

- If your design contains statements between `translate_on` and `translate_off` pragmas that need to be synthesized for emulation, the **Ignore Translate_On/Off** checkbox should be selected (it is cleared by default).
- When some source files of different types (Verilog and SystemVerilog or Verilog and VHDL) have been declared without using the appropriate file type, the **Auto Language Chooser** box can be selected so that the synthesizer automatically analyzes the files for the appropriate syntax. The file type selection is done on the file extension. This option is cleared by default.
- The **Verilog Language** and **VHDL Language** selectors are recommended when the RTL source files are not compatible with the default settings:
 - Verilog-2001 is the default for Verilog files. Other possible choices are Verilog-95 and SystemVerilog if source files are not compatible with Verilog-2001.
 - VHDL-93 is the default for VHDL files. Other possible choice is VHDL-87 if source files are not compatible with VHDL-93.

Advanced commands for the RTL Front-End can be declared in a separate file which can be taken into account for compilation through the **Add Command Files...** item of the contextual menu of the RTL group.

Additional settings can be modified in the other tabs of the **RTL Group Properties** panel:

- The **Synthesizer** tab includes additional options for the synthesizer, in particular the **Synthesis Mode** selectors, the **No. of FPGAs** selector and the **Optimization vs Debugging Facilities** frame:

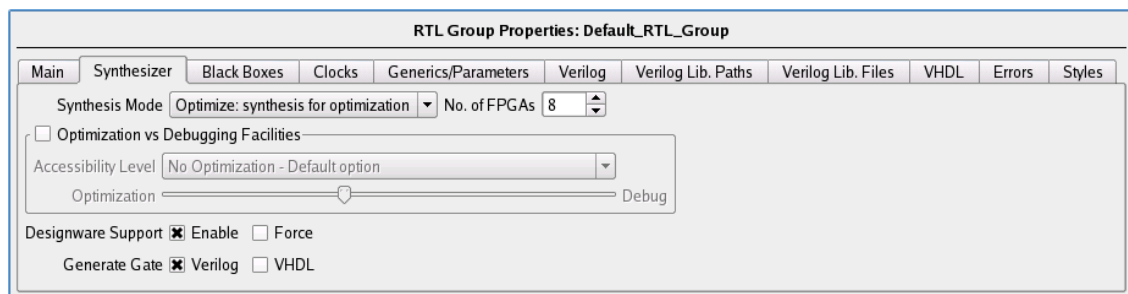


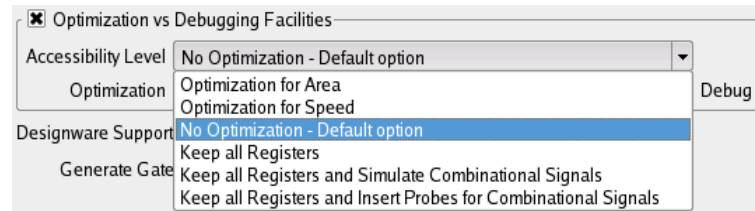
Figure 22: RTL Group Properties → Synthesizer tab

Optimize is the default value for the **Synthesis Mode** selector because it is the most appropriate mode to start with.

The **Block-based** mode is recommended for improved debugging capability and the **Top-Down** mode should be used only for very small designs (less than 2 FPGAs). The **Quick** mode provides faster synthesis, with similar optimizations as **Optimize** mode.

The **No. of FPGAs** selector must be set by user with regard to the ZeBu-Server system which is targeted. The default value is 8, and is not automatically modified when selecting a configuration file for the compilation.

The **Optimization vs Debugging Facilities** frame offers the capability to balance the use of logic resources with easy runtime access to internal signals of the design, and should be set considering the synthesis mode previously set:



The possible values in the **Accessibility Level** selector are the following ones:

- **Optimization for Area:** maximum optimization; decreases register accessibility. Recommended synthesis modes: **Optimize** or **Quick**.
 - **Optimization for Speed:** intermediate optimization; may decrease register accessibility. Recommended synthesis modes: **Optimize** or **Quick**.
 - **No optimization - Default option:** may remove unused registers. Recommended synthesis mode: **Quick**.
 - **Keep all Registers:** all registers are accessible; combinational signals are not accessible, except if dynamic probes are inserted manually in a RTL-based compilation script. Recommended synthesis mode: **Block-based**.
 - **Keep All Registers and Simulate Combinational Signals:** all signals are accessible; the value of combinational signals will be calculated from register values. Recommended synthesis mode: **Block-based**.
 - **Keep All Registers and Insert Probes for Combinational Signals:** all signals are accessible; probes will be added for combinational signals, which will increase the effective size of the design. Recommended synthesis mode: **Block-based**.
- If some modules of the design are known to be blackboxes, they should be declared in the **Blackboxes** tab instead of being created as empty modules in an RTL source file:

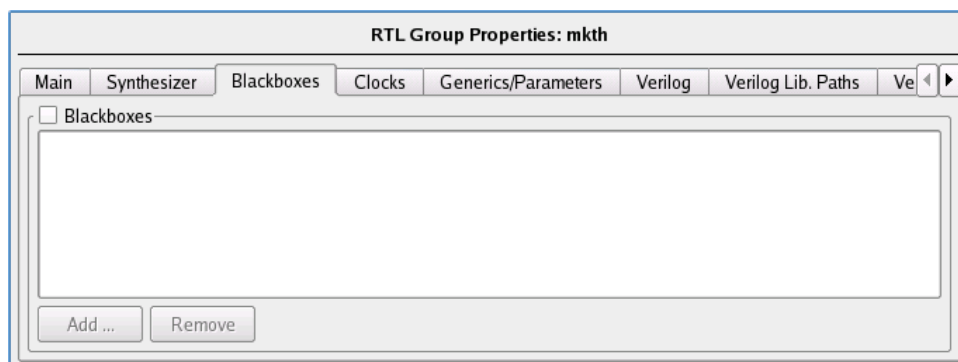


Figure 23: RTL Group Properties → Blackboxes tab

Blackboxes should be considered also for the back-end compilation, as described in Section 3.7.7.

- Gated clocks and generated clocks should be declared in the **Clocks** tab so that the synthesizer optimizes them, as described in Section 3.5.4:

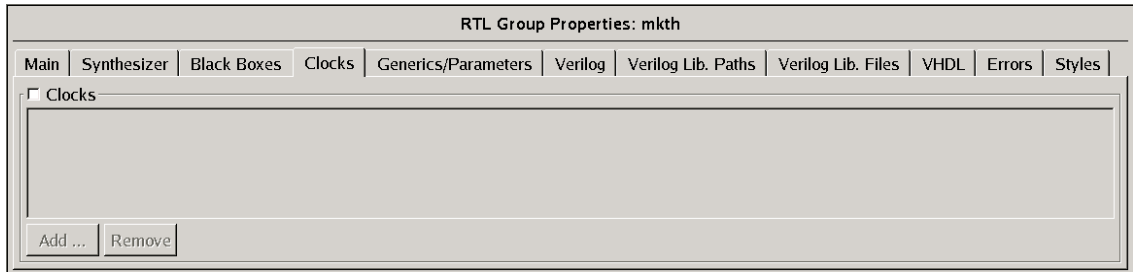


Figure 24: RTL Group Properties → Clocks tab

- The other tabs (**Generics/Parameters**, **Verilog**, **Verilog Lib. Paths**, **Verilog Lib. Files**, **VHDL**) match the usual settings of most simulators.

3.3 EDIF Source Files

Although the ZeBu compiler includes the appropriate tools to synthesize RTL source files, it is also possible to use EDIF netlists that would have been synthesized out of the ZeBu compilation flow. It is recommended to use an FPGA synthesizer but it is also possible to retarget the design if it has been synthesized with an ASIC synthesizer, as described in Section 5.1.

3.3.1 Declaring EDIF Source Files in zCui

To declare EDIF source files for your ZeBu compilation project, you have to right-click on the **EDIF Sources** item of the Project tree and select **Add Edif Sources...**



Figure 25: Contextual Menu to add EDIF Source Files

With the **Files of Type** selector of the file selector, you can choose EDIF netlists (*.edf, *.edif, *.edn) or compressed EDIF netlists (*.edf.gz, *.edif.gz, *.edn.gz) as design files instead of RTL files.

Once the EDIF source file has been added to the project, you can right-click on the file name in the **Project Tree** pane for additional options, in particular to remove the file:

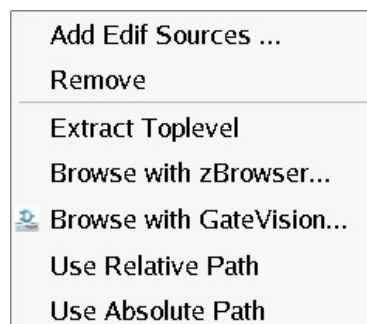


Figure 26: Contextual Menu for an EDIF source file



3.3.2 Integrating DesignWare Library Components with ZeBu

Most ASIC designs use Synopsys' DesignWare components which are not readily available for an FPGA implementation. Such components are automatically mapped to ZeBu when synthesizing with Synplify Premier.

With **zFAST** and other third-party synthesizers (such as Synopsys's Synplify or Synplify Pro and Xilinx's ISE XST), EVE's ZeBuWare (ZW-FPGA) is a specific package containing components which are fully compatible with Synopsys' DesignWare library. Specific recommendations for ZW-FPGA are available in a dedicated Application Note: *[ZW-FPGA Usage - Mapping Synopsys' DesignWare® Components on ZeBu](#)* (AN022).



3.4 Verification Environment

The verification environment, in particular the communication between the design under test (DUT) and the testbench(es), impact the compilation process, in particular by changing the inputs that the user provides.

The most popular environments are described in the present section:

- Cycle-based emulation
- Connecting a software debugger
- Integrating an EVE transactor
- ZEMI-3 transactor
- Debugging with the SRAM trace

For a better flexibility when integrating several transactors, it is possible to declare in the compilation project which FPGA modules of the ZeBu-Server system are actually used to map the transactors for runtime, as described in Section 3.4.10.

The recommendations to integrate a hardware environment with ZeBu (in particular through the Direct ICE interface) and the development of message-based transactors based on the zcei interface is not described in the present manual.

3.4.1 Design Verification Environment (DVE) file

A ZeBu proprietary file, known as the Design Verification Environment file (DVE file) describes the connection between the design and the testbench. This file is a mandatory input for the ZeBu compiler.

3.4.1.1 Introduction

The DVE file is generally written manually. It requires a good knowledge of the design, of the verification environment and of the way to declare the appropriate information. A template can be generated by the ZeBu compiler for HDL/SystemC co-simulation, C/C++ co-simulation or SRAM trace. It is easy to re-use a DVE file or pieces of a DVE file for multiple similar designs or interfaces.

The primary clocks of the design which are connected to the ZeBu clock generator are also declared in the DVE file, as described in Section 3.4.10. The syntax for the DVE file is similar to Verilog syntax.

The declaration of the DUT is implicit in the DVE file, through the declaration of its connections to the driver(s) and/or monitor(s): matching is done on the names of the DUT ports and signals as they exist in the design.

By default, the paths in the DVE file are regarded as EDIF paths. In such cases, some RTL signal names may not be available in the generated EDIF netlists if the **Optimization vs Debugging Facilities** option for synthesis is set to maximum optimization (see Section 3.2.4.1 and 3.2.4.2).



When the design is synthesized with **zFAST**, it is possible to declare RTL hierarchical paths in the DVE file by adding the following line in it:

```
defparam rtlname = yes;
```

Where **yes** can be replaced by **true**.

Notes:

- RTL paths are not supported for elements such as **zceiClockPort**, **zClockPort** and **zIceClockPort**.
- RTL path expressions are Verilog compliant.
- When a Verilog path includes special characters, the DVE file must be written with escaped characters.

Detailed information about the content of the DVE file will be described in the [ZeBu-Server Reference Manual](#) (available with a future software release).

3.4.1.2 Other elements of the DVE file

In addition to the instantiation of the hardware and software drivers/monitors, the DVE file also includes the following elements:

- Declaration of the static and dynamic triggers. You can have up to 16 triggers (whatever their types, without any specific limitation for each type).
- Insertion of registers for runtime access to a DUT input or to an undriven internal signal (for example, a reset input pin).
- Declaration of **assign** statements which may be used to force DUT input signals, to create a top-level port for an internal signal of the design or to interconnect some clocks.
- Instantiation of Verilog logic gates (AND, OR, NAND, XOR, BUF, etc.) which may be useful for the connection of the design to a specific environment.

The following advanced features can be declared in the DVE file:

- Optional frequency constraints for the system clocks of ZeBu.
- Optional frequency constraints for the primary clocks of the design.
- Connection of the synthesized clocks, including their frequency constraints, as described in Section 5.2.1.2.

3.4.1.3 Generating a DVE file template with **zCui**

For cycle-based emulation, **zCui** can generate a DVE file template which includes:

- The instantiation of the appropriate co-simulation driver (**HDL_COSIM**, **C_COSIM** or **MCKC_COSIM** drivers described in Section 3.4.2). The DUT top ports are connected according to their direction at the DUT interface.
- When some clock signals have been declared in the **Clock Declaration** panel as described in Section 3.5.2, the **Controlled Clocks** are added in the DVE file for connection to the ZeBu clock generator. If no clock is declared in the **Clock Declaration** panel, a controlled clock is automatically added in the DVE file but it is not connected to the design.

Note that the automatic generation of the DVE file is also possible when targeting SRAM trace, as described in Section 3.4.6. In such a case, the primary clocks of the

design must be declared in the **Clock Declaration** panel and all the top-level ports of the design are connected to the SRAM trace monitor.

The settings for the generation of the DVE file template are available in the **Environment** panel. In the **DVE** tab, you should select **Auto Generated** in the **Generation Mode** frame and then select the appropriate driver in the **Generator Type** frame.

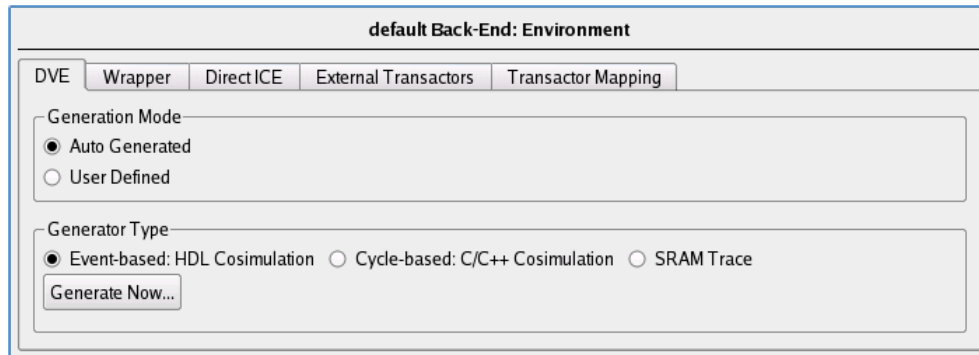


Figure 27: Environment → DVE tab for Automatic Generation

You can either wait until the ZeBu compilation is launched or click **Generate Now**. When clicking **Generate Now**, the synthesis is launched if necessary and EDIF sources as well as clock declarations in **zCui** are checked.

When your emulation environment includes some additional drivers that cannot be added automatically in the DVE file template, it is recommended to click **Generate Now** then switch to **User Defined** to finalize the content of the DVE file before launching the ZeBu compilation (the file resulting from the automatic generation is the one proposed by default in the **DVE File Name** field).

3.4.2 Hardware and Software Co-Simulation

For co-simulation, the ZeBu package includes the following drivers:

- For C/C++ co-simulation:
 - Cycle-based, bit-accurate C/C++ co-simulation: C_COSIM
 - Event-driven, bit-accurate C/C++ co-simulation: MCKC_COSIM
- For HDL co-simulation: HDL_COSIM
- For SystemC co-simulation, with an event-driven bit-accurate SystemC testbench: HDL_COSIM (the co-simulation wrapper will be different from the one generated for HDL co-simulation).

Advanced information about these verification modes is available in the dedicated manuals:



- The *ZeBu HDL Co-Simulation Manual* describes the specific rules for instantiating the HDL_COSIM driver.
- The *ZeBu C++ Co-Simulation Manual* describes the choice criteria between C_COSIM and MCKC_COSIM drivers and the specific rules for instantiating each of them.

3.4.3 Integrating a Synthesizable Testbench

ZeBu offers the capability to synthesize and emulate a testbench to control and monitor the Design Under Test (DUT). Both the DUT and the testbench are mapped in the ZeBu system.

ZeBu handles the synthesized testbench as any other driver/monitor which is declared in the DVE file.

3.4.4 Connecting to a Software Debugger

3.4.4.1 Connecting through the Smart Z-ICE Interface

The Smart Z-ICE interface (64 data pins and 4 clock pins on a 2-slot unit; 80 data pins and 5 clock pins on a 5-slot unit) provides the support of standard software debuggers such as JTAG debuggers.

When the design is connected to a software debugger through the Smart Z-ICE interface, the DVE file includes the connection to the appropriate pins of the Smart Z-ICE interface through the SMART_ZICE_ZSE driver.

The *ZeBu-Server Smart Z-ICE Manual* includes detailed information about hardware connection and software integration.

3.4.4.2 Connecting through the JTAG Transactor

The EVE Vertical Solutions' catalog includes a JTAG transactor which can be used to connect to a software debugger without any additional hardware.

For details about the ZeBu JTAG Transactor, please contact your usual EVE representative.

3.4.5 Integrating a Transactor from EVE Vertical Solutions' Catalog

When integrating a ZeBu Transactor developed by EVE Vertical Solutions, some additional parameters should be set. A User Manual is provided for each ZeBu transactor provided by EVE.

- Set the \$ZEBU_IP_ROOT environment variable before launching **zCui**.
- In the **Back-End** → **Environment** → **External Transactors** tab, select the **Paths to Transactor Models** and the **Use ZEBU_IP_ROOT** checkboxes:

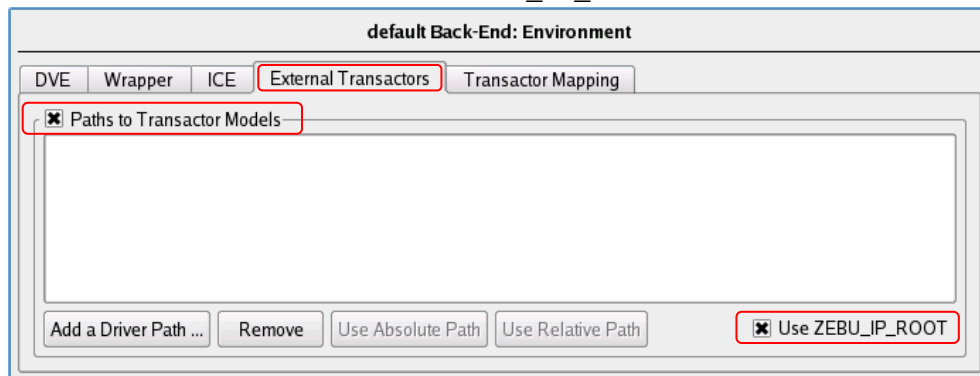


Figure 28: Environment → External Transactors tab for an EVE Transactor

- In the DVE file, connect the interface of the design to the transactor by instantiating the driver as it is recommended in the transactor's User Manual.
- Several instantiations of the same or different drivers can exist in the same DVE file in case of a multi-transactor environment.

3.4.6 Integrating a ZEMI-3 Transactor

The compilation of a ZEMI-3 transactor is fully integrated in **zCui**:

- Create a **ZEMI-3 Transactor** item in the **Sources** item of the **Project Tree** by selecting **Add Zemi-3 Transactor...** in the contextual menu:

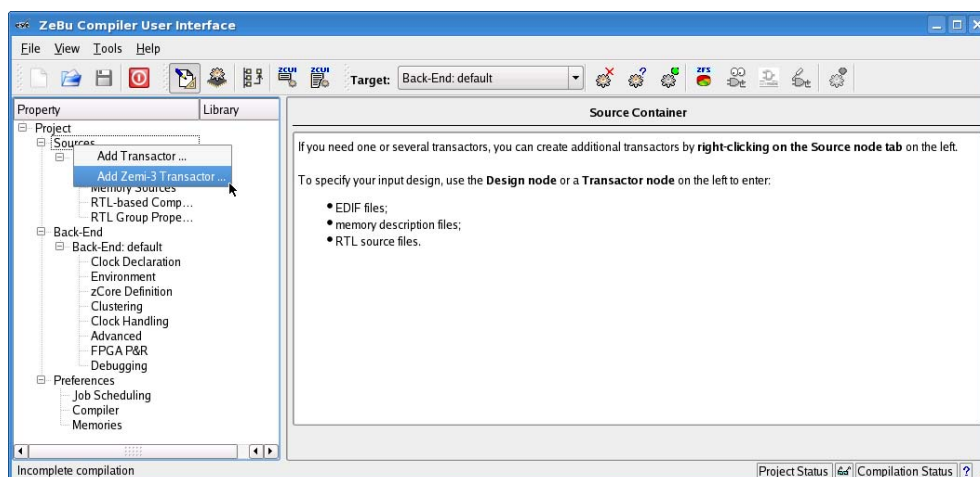


Figure 29: Creating a ZEMI-3 Transactor

- Add the SystemVerilog or Verilog source files and declare the top name of the hardware part of the transactor:

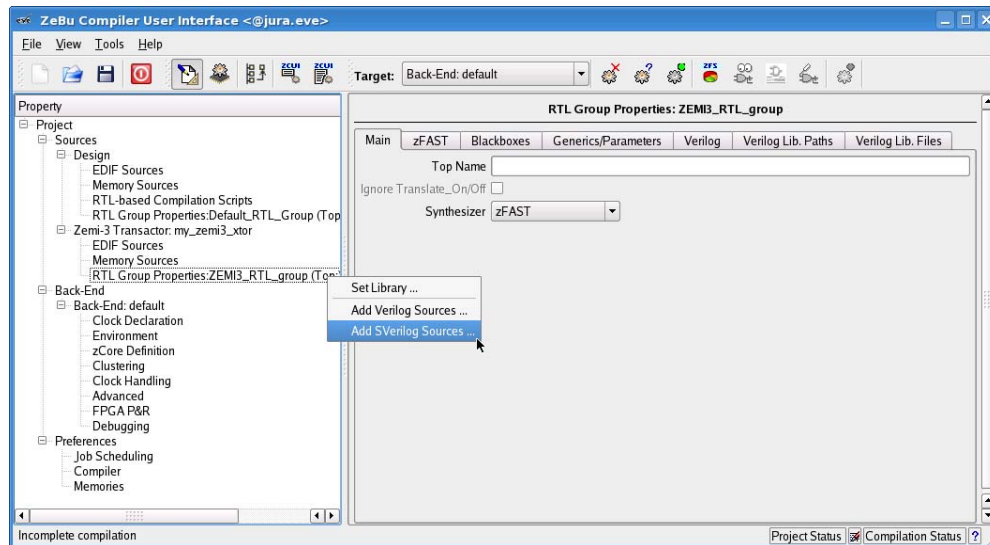


Figure 30: Adding Source Files for a ZEMI-3 Transactor

A ZEMI-3 transactor is instantiated in the DVE file in a similar way to other transactors and co-simulation drivers, except for the clock declaration which is slightly different.

In a ZEMI-3 transactor, the clock is declared as any other port of the transactor. Since the clock has to be a controlled clock generated by ZeBu, the clock port of the transactor is connected to the output of a `zceiClockPort`.

Detailed information about development and integration of a ZEMI-3 transactor is available in the [ZEMI-3 Manual](#).

3.4.7 SRAM Trace

ZeBu-Server includes an internal trace memory in which up to 4,096 signals of the design can be dumped. The ZeBu software includes a dedicated driver which provides this SRAM trace feature. Any interface or internal signal of the design can be connected to the SRAM trace driver.

By default, 256 MBytes of memory are reserved for trace purposes. Depending on the type of modules which populate the ZeBu-Server system, the reserved amount of memory can be increased to 2 or 4 GBytes by modifying the setup of the ZeBu-Server system, as described in the [ZeBu-Server Installation Manual](#) (Rev E).

In the DVE file, the SRAM trace driver (`SRAM_TRACE`) is instantiated with connection of all the signals as output ports (`output_bin`), whatever their actual direction, in particular for top-level ports of the design (the example in Section 3.4.9 shows such connection for SRAM trace).

When targeting SRAM trace, it is possible to generate automatically a DVE file template from `zCui` as described in Section 3.4.1.3. In such a case, the primary clocks of the design must be declared in the **Clock Declaration** panel and all the top-level ports of the design are connected to the SRAM trace monitor. Note that the input



ports of the design will be connected to ground if the DVE file template is not modified afterwards to drive them.

3.4.8 Mixed Environments

The design can be simultaneously connected to multiple hardware and software drivers and monitors. This enables you to achieve the best communication speed between the software and/or hardware testbench and the design, while simultaneously providing full control of the debugging features.

In such a case, the DVE file allows the instantiation of multiple hardware and software drivers.

3.4.9 Example of DVE File

This section presents an example of verification environment which integrates various elements. The DUT communicates with the following:

- 128 Mb ZSDRAM Memory Model (4Mx32).
- ZeBu Video Transactor used in 10-bit RGB mode.
- JTAG Debugger connected through the Smart Z-ICE interface (port 0).
- SRAM Trace for the interface between the DUT and the ZSDRAM memory model.
- 64-bit CRC is implemented in the ZeBu system as an embedded synthesizable testbench.
- A trigger is placed on the control signals of the ZSDRAM memory model to detect a write cycle which corresponds to the following equation:
 $Cs_n==0 \ \& \ Ras_n==1 \ \& \ Cas_n==0 \ \& \ We_n==0$
- The `loop_in` pin of the DUT is connected to a signal driven by the DUT, `loop_out`.

The Video transactor is connected as an instantiation of the `video_driver`; the transactor clock is connected to a `zceiClockPort` and a register is connected to the transactor reset for manual control at runtime. The `video_display` signal of the DUT is forced to 1 for permanent activation of the video features.

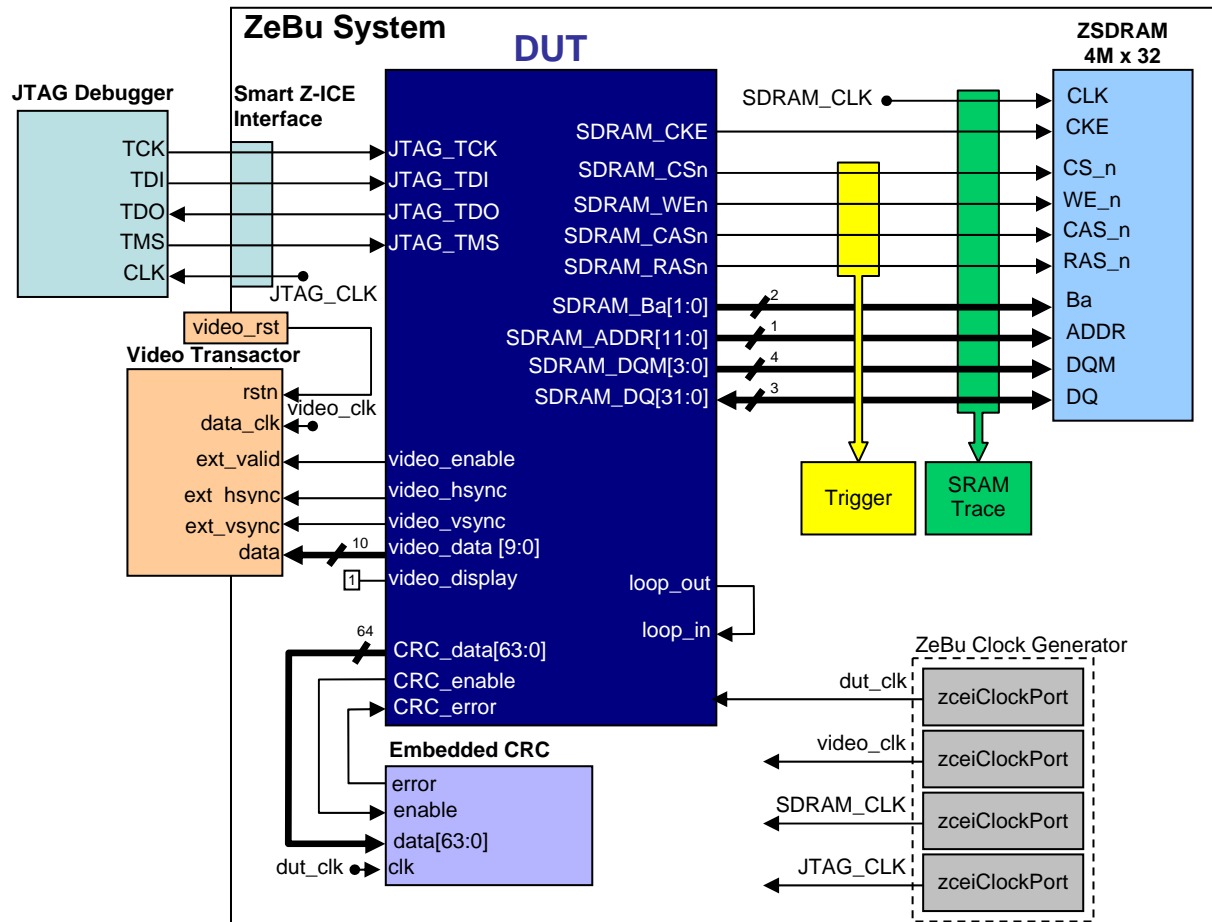


Figure 31: System Architecture for DVE Example

```
// Connection of the DUT clock
zceiClockPort clock_dut(.cclock( dut_clk ));

// Video Transactor (video_driver) instantiation
// with video_clk connected to a clock port
video_driver my_video (
    .rstn (video_rstn),
    .data ({20'b0,video_data[9:0]}),
    .data_clk (video_clk),
    .ext_vsync (video_vsync),
    .ext_hsync (video_hsync),
    .ext_field (1'b0),
    .ext_valid (video_enable)
);

defparam my_video.clock_ctrl = video_clk;
reg video_rst;
assign video_display = 1'b1;

zceiClockPort clock_driver_video (
    .cclock(video_clk),
);
```



```
// Connection of the JTAG Debugger to the Smart Z-ICE interface
// with connection of JTAG_CLK as an output clock from the ZeBu clock generator
SMART_ZICE_ZSE my_Jtag (
    .input_port0_42(JTAG_TCK),
    .input_port0_41(JTAG_TDI),
    .output_port0_40(JTAG_TDO),
    .input_port0_39(JTAG_TMS),
);

defparam my_Jtag.vcc = "1V8";

defparam my_Jtag.clock_out_port0_49=JTAG_CLK;
zceiClockPort clock_driver_smartzice (.cclock(JTAG_CLK));

// Connection of the ZSDRAM clock to a clock port
zceiClockPort clock_driver_ZSDRAM (.cclock(ZSDRAM_CLK));

// Connection of the Trace driver to the ZSDRAM memory interface
SRAM_TRACE my_Trace_on_ZSDRAM (
    .output_bin( {
        SDRAM_CS, SDRAM_WE, SDRAM_CAS, SDRAM_RAS,
        SDRAM_ADDR[11:0], SDRAM_DQ[31:0],
        SDRAM_DQM[3:0], SDRAM_Ba[1:0]}
    )
);

// Set Trigger parameters
trigger SDRAM_WriteCycle = ((SDRAM_WE == 1'b0) & (SDRAM_CS == 1'b0) & (SDRAM_RAS==
1'b1) & (SDRAM_CAS == 1'b0));

// Declaration of the Synthesizable Test Bench for CRC
CRC myCrc (
    .error( CRC_error ),
    .enable( CRC_enable ),
    .data( CRC_data[63:0] ),
    .clk( dut_clk )
);

// Loop Connection on the DUT interface
assign loop_in = loop_out;
```

3.4.10 Transactor Mapping

When the number of transactors is too high to achieve reasonable runtime performance or when the IF FPGA of FPGA module U0_M0 cannot be compiled correctly, the ZeBu compiler supports a user-defined mapping of the transactors. This is applicable to any user-developed transactor, either zcei-based or ZEMI-3-based, and to transactors from EVE Vertical Solutions' catalog; any other elements instantiated in the DVE are not supported in any FPGA modules other than U0_M0 (SRAM_Trace driver, co-simulation drivers, static triggers, Verilog gates or “DVE registers”).

For that purpose, a dedicated file should be declared in the **Back-end → Environment → Transactor Mapping** tab:

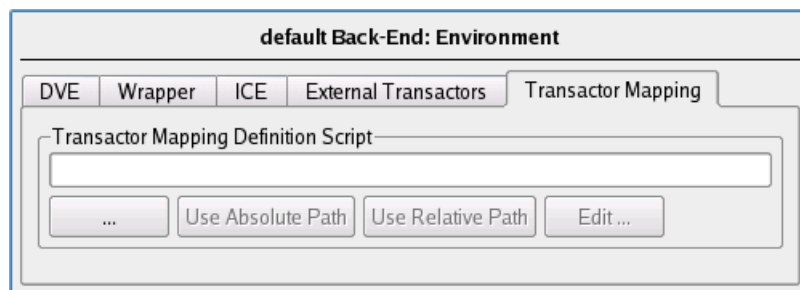


Figure 32: *Environment → Transactor Mapping* tab

The **Transactor Mapping Definition Script** is a Tcl file, composed of the following commands:

```
defmapping <xtor_name> <module_loc>
```

Where <xtor_name> is the name of the transactor as it is instantiated in the DVE file and <module_loc> is the identifier of the module where the transactor is mapped. The syntax for <module_loc> is Ux_My.

Any transactor instantiated in the DVE but not listed in the **Transactor Mapping File** is automatically mapped onto U0_M0.

It is possible to map any transactor instantiated in the DVE file on other modules of U0 unit; in case of a multi-unit system, other units cannot be used.

Example:

```
defmapping counterDriver0 U0_M1 ;
defmapping clockNoise0    U0_M2 ;
```

The following limitations are to be taken into account:

- It is not possible to connect directly transactors mapped on different modules.
- Each input or inout port of the DUT can only be connected to the IF FPGA of one module, although several transactors can be mapped on this module.
- It is not possible to enable/disable a transactor when it is mapped on a module different from U0_M0.

3.5 Clock Modeling

3.5.1 Definitions

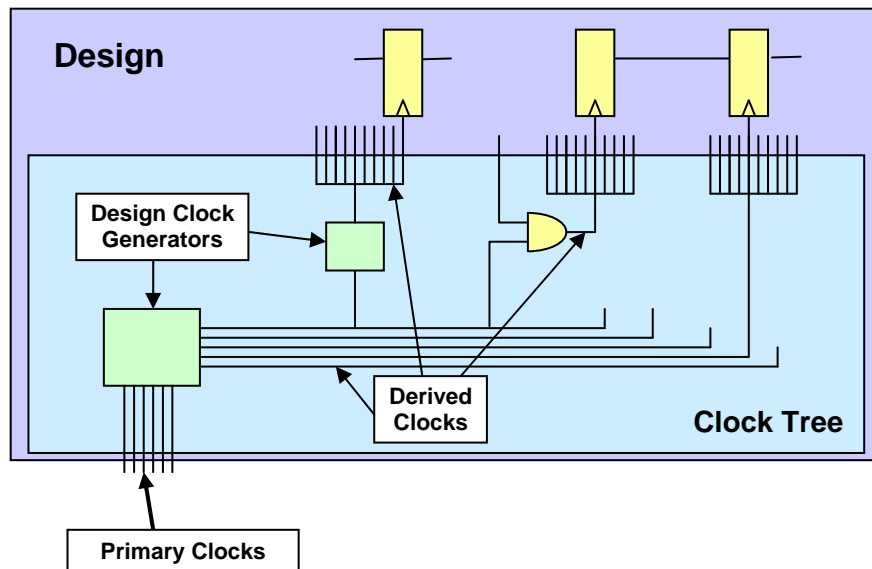


Figure 33: Design Clock Tree

Primary clocks are the external input clocks of the design.

Derived Clocks are the clocks produced from the primary clocks through layers of logic implemented by the clock tree. Derived clocks are also known as secondary clocks.

The **clock tree** includes all the logic between the primary clocks and the clock input connectors of the registers and memories in the design. It typically includes one or several clock generators and individual gates. Starting from the primary clocks, the design clock tree is most often composed of the following elements:

- Global Design Clock Generator: clock frequency multiplication/division for frequency adaptation, clock multiplexing for multimode operation or global gated clock for chip-level power reduction.
- Secondary Design Clock Generators: module clock frequency division for module-level frequency adaptation or module gated clocks for module-level power reduction.
- Individual Gates: latches for synchronization, flip-flops for frequency division or gated clocks for block-level power reduction.

Clock Tree Example:

The following figure shows the clock tree example that will be used further in this chapter to present the different clock mapping solutions.

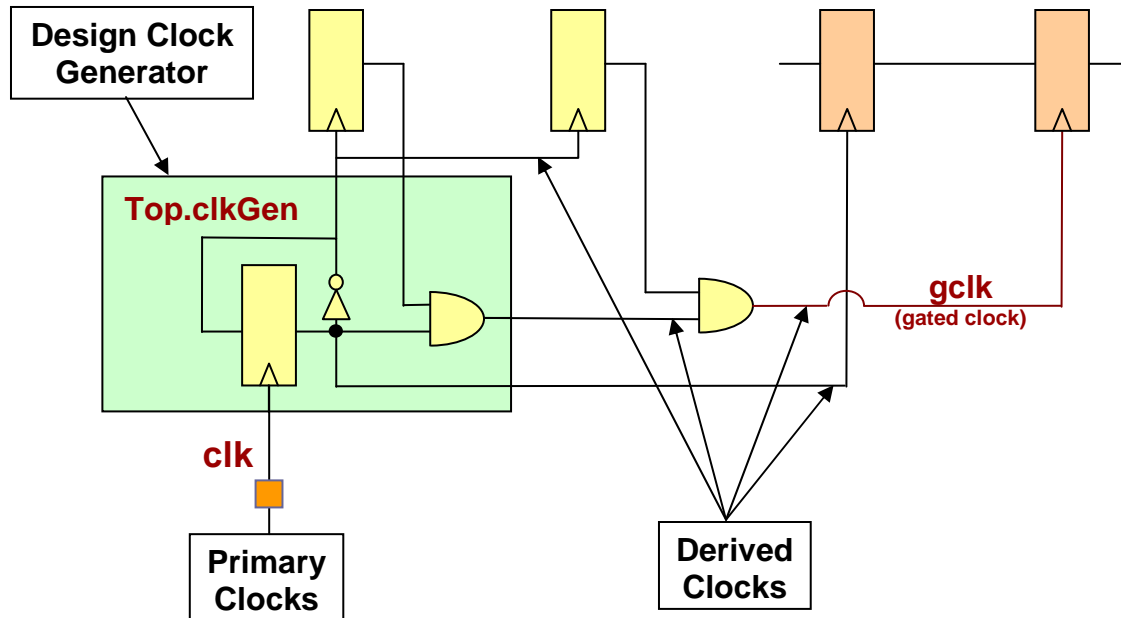


Figure 34: Clock Tree Example

3.5.2 Declaring clocks

The primary clocks of the design should be declared in the **Back-End → Clock Declaration** panel.

When the DVE file is written manually, the primary clocks of the design have to be connected to the DUT inputs as described in Section 3.5.

When generating a DVE file template as described in Section 3.4.1.3, this declaration is mandatory to have the appropriate information in the DVE file.

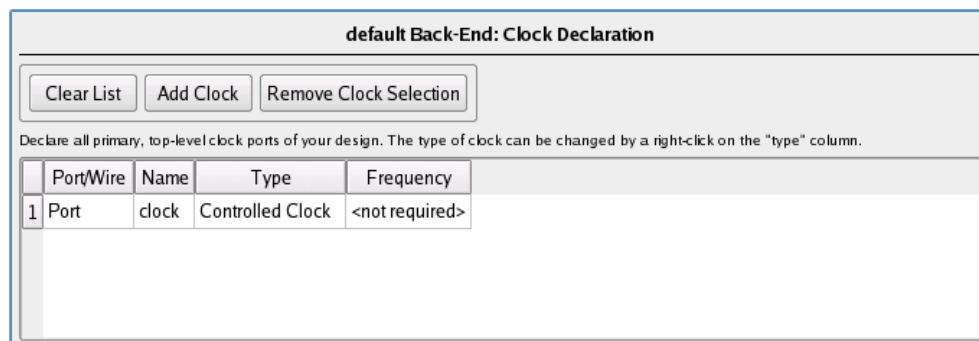


Figure 35: Back-End → Clock Declaration Panel

The **Clock Declaration** panel also supports the following advanced options by changing the value in the **Type** column (selector available with a double click on the **Type** cell for each clock):

	Port/Wire	Name	Type	Frequency
1	Port	CLK	Controlled Clock	not required

Controlled Clock
Other Primary Clock
Synthesized Clock
Internal Clock Path
Internal Data Path

- Declaration of the primary clocks of the design that are not connected to the ZeBu Clock Generator (**Other Primary Clock**).
- Declaration of internal signals of the design as clocks (**Internal Clock Path**) or data (**Internal Data Path**) for advanced clock mapping.
- Declaration of synthesized clocks (**Synthesized Clock**) with their frequency, as described in Section 5.2.1.2.

3.5.3 Clock Modeling for Primary Clocks

In ZeBu-Server, the interface FPGA (IF) implements the ZeBu Clock Generator which distributes up to 16 controlled primary input clocks. Such clocks are mapped automatically by the ZeBu compiler, including the routing on the low-skew network at the system level and in the FPGAs. In the FPGAs, clock buffers (BUFG) are automatically added by the ZeBu compiler for this purpose.

When the design has more than 16 primary clocks or when some primary clocks come from a Direct ICE target or from a software debugger connected to the Smart Z-ICE interface, the additional primary clocks are also distributed via the interface FPGA though they are not generated by the ZeBu Clock Generator.

3.5.3.1 Connecting Primary Clocks to the ZeBu Clock Generator

Up to 16 primary clocks can be connected to the ZeBu Clock Generator.

The connections of a primary clock to the ZeBu Clock Generator are declared as instantiations of `zceiClockPort` macros in the DVE file:

```
zceiClockPort <my_ClockPort> (.cclock(<my_prim_clock>));
```

When a DVE file template is generated as described in Section 3.4.1.3, the primary clocks must be declared as **Controlled Clocks** in the **Clock Declaration** panel (see Section 3.5.2). These clocks are automatically taken into account.

The ZeBu compiler optimizes the resources allocated for the Clock Generator: depending on the number of declared primary clocks, the actual Clock Generator has 2, 4, 8 or 16 outputs.



For runtime flexibility, it may be useful that the ZeBu compiler allocates a larger Clock Generator (in particular to add a clock for runtime monitoring). For that purpose, the following line should be added in the DVE file:

```
defparam fk_zceiClockPort_number=<nb_clocks>;
```

Where <nb_clocks> is an integer which matches the total number of controlled clocks for runtime (zceiClockPort instantiations + additional clocks declared at runtime).

Example:

If the design has 8 primary clocks connected to the ZeBu Clock Generator, and 1 additional clock is needed for trace purposes at runtime, the following line should be added in the DVE file:

```
defparam fk_zceiClockPort_number=9;
```

3.5.3.2 Additional Primary Clocks

When the design includes more than 16 different primary clocks, it is recommended to connect to the ZeBu Clock Generator the 16 clocks which have the highest fan-out. The additional primary clocks are declared as **Other Primary Clocks** in the **Clock Declaration** panel for an optimized routing in ZeBu.

Some specific recommendations are available in Section 5.2.1 for asynchronous primary clocks which can be clocks driven by the Direct ICE or Smart Z-ICE interfaces or fast synthesized clocks for prototyping.

3.5.4 **Clock Modeling for Derived Clocks**

In order to optimize the ZeBu compilation process and the achievable runtime performance, it is recommended to simplify the clock tree as much as possible in the RTL code by forcing the test clocks or merging the clocks which have the same runtime behavior.

In addition, the Synplify synthesizers with **zRtlFrontEnd** offer options to optimize the gated clocks during synthesis if these clocks are declared in the **Clocks** tab for the corresponding RTL Group, as shown in Figure 24.

Once the clock tree of the design has been optimized, the mapping of derived clocks may still generate some clock skew and glitches. The ZeBu compiler provides automatic processing which fixes these drawbacks for derived clocks as described in Section 3.5.4.1.

3.5.4.1 Automatic Mapping of Derived Clocks

The ZeBu compiler automatically identifies the derived clocks from the declaration of primary clocks in the DVE file. Based on this identification, the ZeBu compiler automatically inserts the appropriate elements to offset the clock skew and remove the glitches in the clock tree.

The default settings of the ZeBu compiler in the **Clock Handling** panel are set for an easy bring-up of the design, when the primary clocks of the design all come from the ZeBu Clock Generator. The following figure shows which are the default settings:

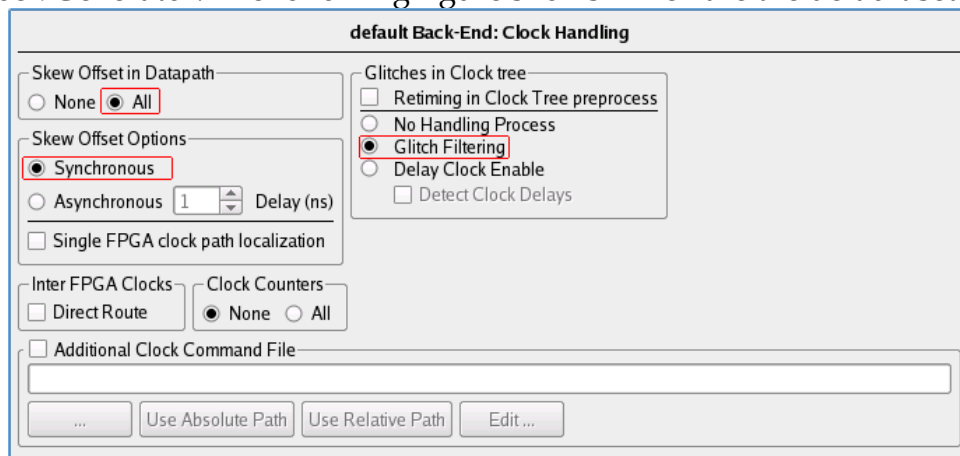


Figure 36: Back-End → Clock Handling panel with default settings

When some primary clocks do not come from the ZeBu Clock Generator (in particular when using asynchronous primary clocks) and when optimizing the runtime performance of the design, the clock handling parameters should be modified as described in Section 5.2.

With the default compilation settings, the parameters estimated during static timing analysis for the control of inserted delays and glitch filters are automatically transmitted to the runtime environment, as described in Section 3.5.5.

3.5.5 Static Timing Analysis

The Static Timing Analysis is always active and can be configured from the **Static Timing Analysis** frame in the **Back-End** → **Configuration** panel:

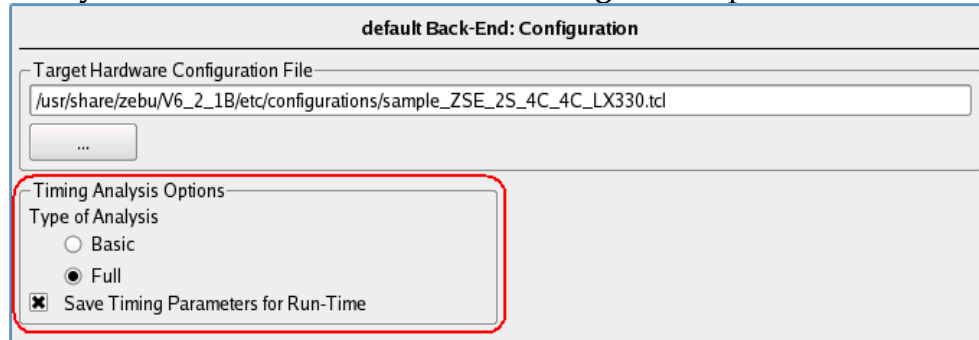


Figure 37: Timing Analysis Options Frame

By default, it takes into account the inter-FPGA combinational paths (**Type of Analysis** is set to **Full**). The driverClock frequency and clock handling parameters estimated during timing analysis are automatically taken into account at runtime since the **Save Timing Parameters for Run-Time** checkbox is selected.

Choosing **Basic** for the type of analysis provides an estimation of the maximum frequency if there were no inter-FPGA combinational paths in the design. If such inter-FPGA combinational paths exist, the reported frequency of the **Basic** mode is probably too optimistic and it might lead to non-functional runtime emulation.

When **Glitch Filtering** is selected in the **Clock Handling** panel (default setting), the static timing analysis takes into account the inserted filters when it estimates the driverClock frequency and the zClockSkewTime and zClockFilterTime parameters.

If a specific value has been declared in the DVE file for the inter-FPGA communication clock, the static timing analysis takes it into account when estimating the achievable runtime frequency.



3.6 Distributing the design in the system

3.6.1 Introduction

Compiling a design for ZeBu includes the distribution of the design among the units, modules and FPGAs of the targeted ZeBu-Server system, considering user-specific size and runtime performance criteria.

The size criteria are mostly applicable for multi-user environments where the Zebu-Server systems have to be shared. The performance criteria are mostly applicable for large, complex designs.

The ZeBu compiler offers the capability to automatically create this distribution or to proceed with manual tuning when attempting to optimize the size or performance.

The various Zebu-Server configurations offer flexibility for the distribution of the design:

- Mapping the design on a multi-unit system, based on 5-slot units, provides the support of large designs with relocation capability in a multi-user environment.
- Mapping the design on several small modules provides flexibility, in particular with relocation capability for small designs in a multi-user environment.
- Mapping the design on few bigger modules, with optional loopback modules in a 5-slot unit, provides better interconnection which leads to higher runtime performance.
- When integrating a hardware target connected through the Direct ICE interface, some physical constraints appear for the memories (less memory resources on the 8C/ICE module than on other modules).

When the design exceeds the size of 1 module and uses more than 8 FPGAs, it is recommended to split the design into several subsets in order to break the back-end compilation into several parallel processes. These subsets are called “zCores”, which can be declared manually or generated automatically by the ZeBu compiler, as described in Section 3.6.2.

A smaller design is processed as a single zCore without any specific declaration by user.

For each zCore, the compiler processes the most critical steps of the back-end compilation, in particular uniquification of the netlists and clock processing. It also maps the design on the FPGAs of the targeted module, either with performance oriented partitioning (POP) or density oriented clustering (DOC), as described in Section 3.6.3.

When the distribution of the design is optimized for performance, the FPGA filling rates are potentially lower in order to take advantage of the multiple data rate communication (xDR) during the system place and route process.

3.6.2 System-level Partitioning

3.6.2.1 Introduction

The system-level compilation splits the EDIF netlist, resulting from synthesis in the ZeBu compiler or declared as source files, into zCores and reserves the necessary FPGAs for each zCore.

The most common use models are the following:

- When a design is small enough to be mapped on only 1 module or is smaller than 16 FPGAs, it is not necessary to declare anything and the design is compiled as a single zCore.
- In some very specific cases, declaring small-sized zCores (1 zCore fitting 1 or 2 FPGAs) may provide a higher runtime frequency for small designs.
- For a large design in which there are no functional blocks which can easily be identified as zCores, it is recommended to use automatic zCore generation as described in Section 3.6.2.2.
- For a large design with few functional blocks identified as zCore candidates such as a CPU core or a graphic processor in a SOC, it is recommended to manually declare these blocs as zCores, as described in Section 3.6.2.2, and use automatic zCore generation for the rest of the design, as described in Section 3.6.2.2.
- For a large design with most of it declared in zCores, the small residual part (“uncored” logic) can be left without automatic zCore generation: it is shared automatically in the user-defined zCores by the system-level compiler or a dedicated small zCore is automatically created.

3.6.2.2 Automatic zCore Generation

The ZeBu compiler can automatically partition the design into several zCores which are each mapped on a module in the ZeBu-Server system. This automatic zCore generation is processed by the system-level compiler, and is supported for designs which do not use the Direct ICE interface.

This feature can be activated from the **Back-End → Clustering** tab:

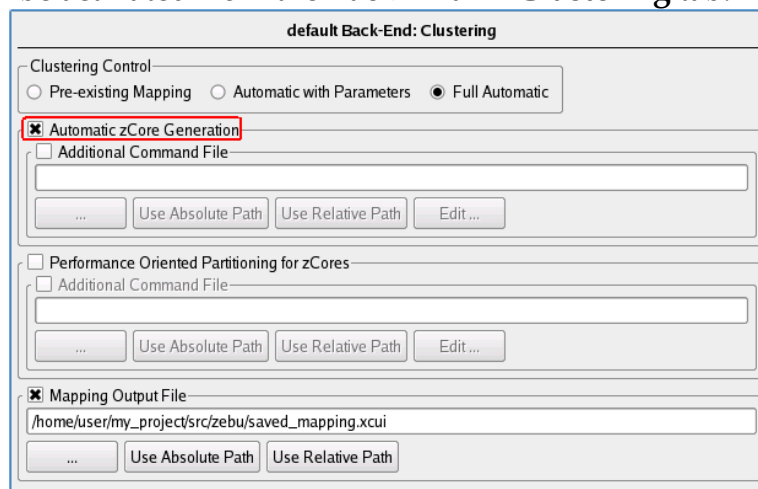


Figure 38: Automatic zCore Generation in the Clustering tab

When the **Automatic zCore Generation** checkbox is selected, the system-level compiler proceeds with the creation of the zCores. The settings of the **Clustering Control** frame are applicable for automatic zCore generation, as described in Section 3.6.4.

It is strongly recommended to declare a **Mapping Output File** that will be used in future compilations when the automatic generation is skipped (when **Pre-existing Mapping** is selected, as described in Section 3.6.4.3).

An **Additional Command File** can be declared to configure the generation process, as described in Section 5.5.2.

3.6.2.3 Manual declaration of the zCores

Defining the zCores manually requires a good knowledge of the design to choose correctly the hierarchies which are gathered. In most cases, the declaration of zCores is based on the functional architecture of the design, for example zCores can be declared for a CPU core or a complex memory controller of the design.

A recommended size for zCores would be between 8 and 16 FPGAs per zCore for large designs or 1 to 2 FPGAs for smaller designs; the reasonable maximum size is ~32 FPGAs for an acceptable duration of the zCore-level compilation (~500 MBytes of RAM and ~2 min per FPGA).

In order to optimize the distribution among the ZeBu-Server system, it is recommended to declare zCores with a size of 1 FPGA module (or, in case of several smaller zCores, to have their added sizes fitting in 1 FPGA module).

Since the system-level compilation allocates a whole number of FPGAs per zCore and does not share any FPGA between several zCores, it is recommended to define large enough zCores to avoid lost resources in design FPGAs. However, if the size of a zCore is very large, it may impact the duration and required resources for the zCore-level compilation.

When some portions of the design are not declared in any zCore, the system-level compilation can either share them in the user-defined zCore (on resources and performance criteria) or process them as a whole in a similar way to other zCores. It is recommended to have the smallest amount of “uncored” logic for better runtime performance.

The zCores are declared through a Tcl file (**zCore Definition File**) in the **Back-End** → **zCore Definition** panel:

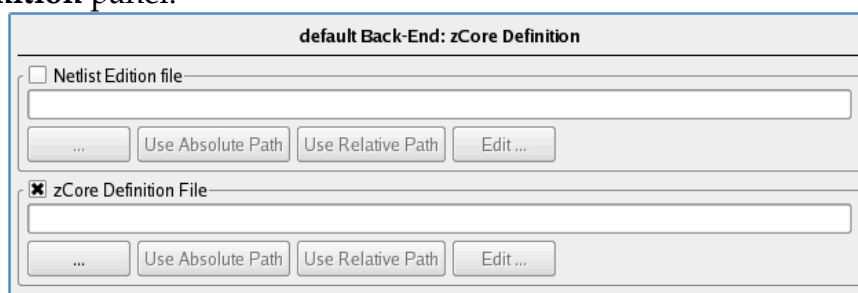


Figure 39: Back-End → zCore Definition panel



By default, the hierarchical paths declared for zCores definition are EDIF paths but, when synthesizing with **zFAST** in block-based mode, RTL paths can be declared as well.

Declaration of the zCores with a list of paths

All the zCores are declared in the Tcl file with the following command:

```
defcore <zcore_name> -path_list <path_list> [-rtlname]
```

Where <zcore_name> is the user-defined name of the zCore, <path_list> is a Tcl list of the hierarchical paths belonging to <zcore_name>; by default, this list includes EDIF paths but the -rtlname option can be used when synthesizing with **zFAST** to declare RTL paths (only supported with block-based synthesis mode).

Example:

In the following example, core0 includes a list of instances and core1 includes only one instance:

```
defcore core0 -path_list { DUTtop.big_block_A DUTtop.big_block_B }  
defcore core1 -path_list DUTtop.decoder
```

Declaration of the zCores with an intermediate file

In some cases, it can be easier to declare all the hierarchical paths in a separate file and use this file when declaring the zCore:

```
defcore <zcore_name> -path_file <zcore_decl_file> [-rtlname]
```

Where <zcore_name> is the user-defined name of the zCore, <zcore_decl_file> is the path to the file in which the hierarchical paths are declared.

The zCore declaration file is a text file in which each line is the hierarchical path to an instance which will be gathered in the zCore. Any line starting with a # sign is ignored. By default, this file includes EDIF paths but the -rtlname option can be used when synthesizing with **zFAST** to declare RTL paths (only supported with block-based synthesis mode).

As for any other file declared in a command, only an absolute path should be used for <zcore_decl_file> in the defcore command.

Example:

```
defcore core2 -path_file  
/home/user/my_project/src/zebu/def_core2.lst
```

Where the content of def_core2.lst is:

```
# This is def_core2.lst file  
DUTtop.big_block_C  
DUTtop.big_block_D
```



3.6.2.4 Assigning FPGA resources

In order to optimize the FPGA allocation, the `use_fpga` command can be added for the system-level compiler (in the **zCore Definition File** (**zCore Definition** panel):

- Directive mode: user chooses and specifies the FPGAs which are allocated, using the `-fpga` option (to allocate a single FPGA):

```
use_fpga -zcore <zcore_name> -fpga <unit>.<module>.<fpga>
```

or the `-module` option (to allocate all the FPGAs of a given module):

```
use_fpga -zcore <zcore_name> -module <unit>.<module>
```

Note that several `use_fpga` commands can be declared for the same zCore with `-fpga` and/or `-module` options to allocate several FPGAs for the same zCore.

- Constraint mode: user declares some constraints for the number of FPGAs (`-number` option) or for authorized/forbidden FPGAs (`-constraint` option):

```
use_fpga -zcore <zcore_name> -number <nb_fpga>
```

or

```
use_fpga -zcore <zcore_name> -constraints <list_of_fpgas>
```

Where `<list_of_fpgas>` contains physical FPGA identifiers.

Note that it is not possible to have both directive and constraint modes commands for the same zCore.

Examples:

The compiler can be configured to use 15 FPGAs:

```
use_fpga -zcore my_zcore1 -number 15
```

It is also possible to configure the compiler to choose all the FPGAs on modules M0 and M1 in unit U0 and exclude FPGA16 from the selected FPGAs on U0.M0:

```
use_fpga -zcore my_zcore2 -constraint {U0.M0 U0.M1 ~U0.M0.F16}
```

3.6.3 zCore-level Partitioning

3.6.3.1 Introduction

For each zCore, the ZeBu compiler automatically processes a mapping solution for the reserved FPGAs based on the settings defined in the **Clustering** panel.

The zCore-level compiler offers 2 different modes:

- Density Oriented Clustering: this mode is recommended when the design fits tightly in the ZeBu architecture, requiring high FPGA filling rates or when the design includes many flexible probes and/or SVAs.
- Performance Oriented Partitioning: the main criterion for this partitioning is the achievable runtime performance and the FPGA filling rate may be slightly lower than with the Density Oriented Clustering. This mode is described in Section 3.6.3.3.

Note that it is possible to use the above modes on a zCore-by-zCore basis: the content of a given zCore may lead to choose a different mode.

3.6.3.2 Density Oriented Clustering

The density oriented clustering is activated by default when **Full Automatic** or **Automatic with Parameters** are selected in the **Clustering Control** frame without **Performance Oriented Partitioning** frame being selected. It is disabled if **Pre-existing mapping** option is selected in the **Clustering Control** frame or if performance oriented partitioning is activated, as described in Section 3.6.3.3.

Some advanced settings can be modified by adding commands in the additional command file declared in the **Advanced** → **Build System Parameters** frame.

3.6.3.3 Performance Oriented Partitioning

It is possible to activate the Performance Oriented Partitioning feature which, at zCore-level, automatically maps the design onto the FPGAs.

This partitioning can be activated from the **Clustering** panel. In such a case, the feature is activated for all the zCores of the design: for user-declared zCores as well as for automatically generated ones.

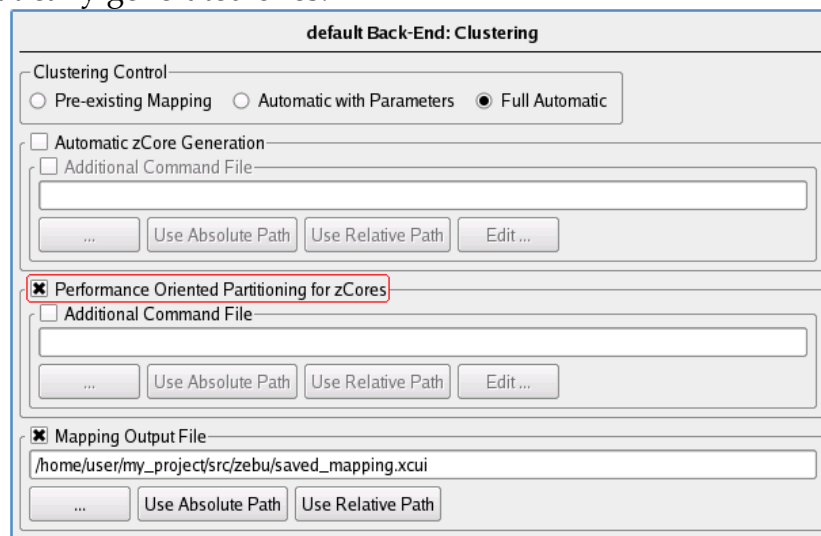


Figure 40: zCore-level Performance Oriented Partitioning

An **Additional Command File** can be declared to configure the generation process, as described in Section 5.5.3.

To use Performance Oriented Partitioning on a zCore-by-zCore basis, the **Performance Oriented Partitioning for zCores** checkbox must not be selected and the following command should be added in the additional command file declared in the **Advanced** → **Build System Parameters** frame:

```
zcorebuild_partitioning -zcore <my_zcore> auto
```

Where <my_zcore> can be set to use performance-oriented partitioning for a given zCore only.

If the `-zcore` option is not present, the automatic performance oriented partitioning is used for all the zCores of the design, as it is when the **Performance Oriented Partitioning for zCores** checkbox is selected.

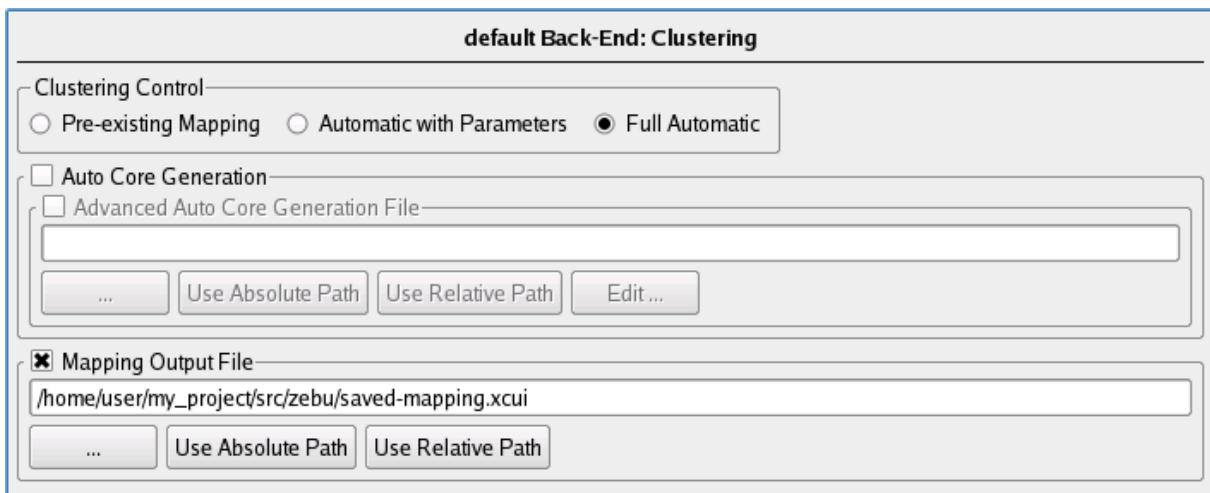
As for automatic zCore generation or Density Oriented Clustering, the Performance Oriented Partitioning uses the settings in the **Clustering Control** frame, as described in Section 3.6.4.

3.6.4 Modifying the settings in the *Clustering* panel

Note that the settings of the **Clustering Control** frame are applicable for both system-level and zCore-level partitioning. In particular, it is not possible to choose **Full Automatic** mode for one and **Pre-existing Mapping** for the other and it is not possible to use different filling rate settings for several zCores.

3.6.4.1 Recommendations for the mapping output file

When **Automatic with Parameters** or **Full Automatic** buttons are selected in the **Clustering Control** frame, it is strongly recommended to declare a **Mapping Output File** in the appropriate field:



The screenshot shows a software interface titled "default Back-End: Clustering". It contains a "Clustering Control" section with three radio buttons: "Pre-existing Mapping", "Automatic with Parameters", and "Full Automatic" (which is selected). Below this is an "Auto Core Generation" section with a checkbox and a text field for "Advanced Auto Core Generation File". At the bottom is a "Mapping Output File" section with a checked checkbox and a text field containing the path "/home/user/my_project/src/zebu/saved-mapping.xcui". Various buttons like "...", "Use Absolute Path", "Use Relative Path", and "Edit ..." are visible next to the text fields.

Figure 41: Clustering panel showing the *Mapping Output File* frame

This file contains all the partitioning constraints: declaration of the zCores, mapping commands in each zCore. It is intended to be an input when switching to **Pre-existing Mapping**. It is a human-readable xml file which may be manually modified for optimization purposes as described in Section 3.6.4.4.

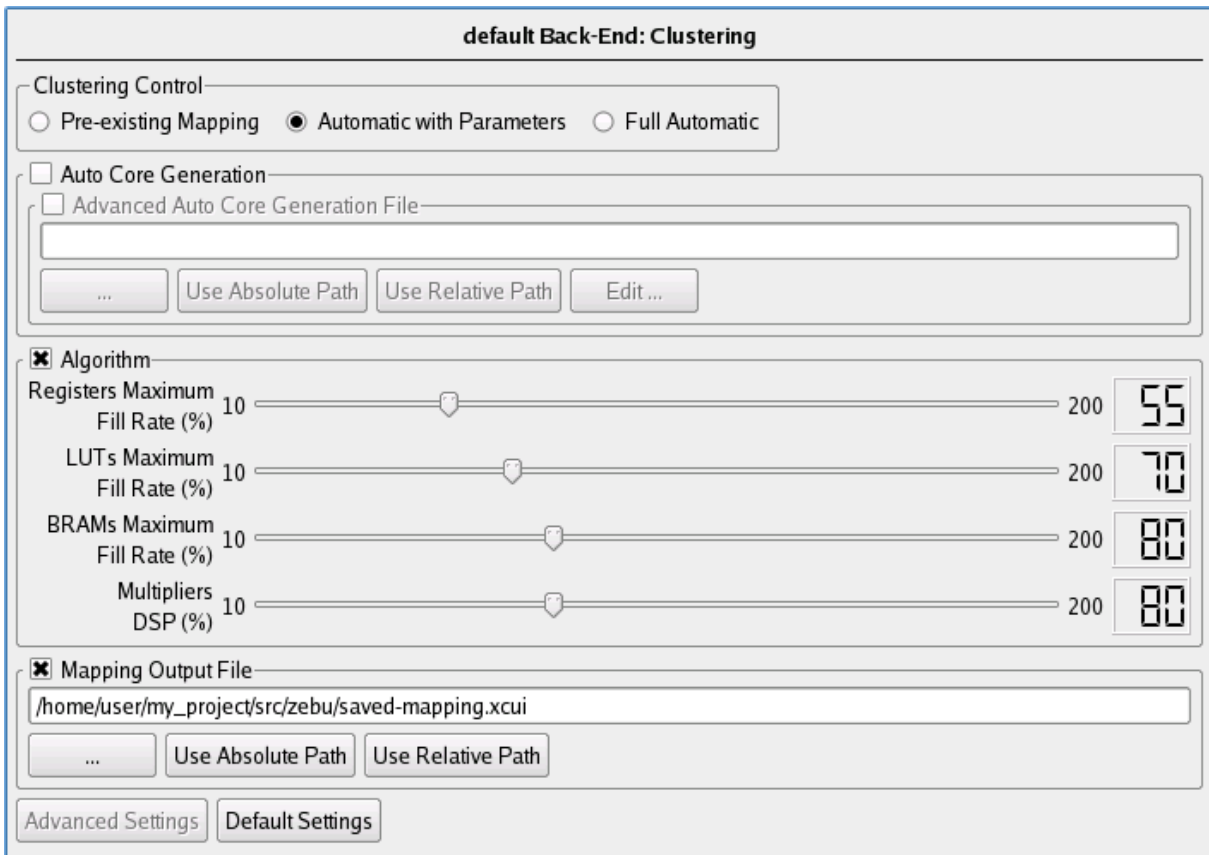
The recommended extension for the mapping file is `.xcui` but there is no constraint on the file name. This file should not be located in the **zCui** working directory to avoid removal in case of a **Clean** operation; it is recommended to store this file with the ZeBu source files of your project (in the `/src/zebu` subdirectory if you used the file tree described in Section 3.1.2).

Note that if the **Mapping Output File** checkbox was not selected when compiling with one of the automatic clustering modes, the mapping file can be created after the successful compilation of the design if there has been no **Clean** operation in the

meantime: proceed in the same way as you would have done before compiling and launch compilation; neither the system-level nor the zCore-level compilation steps are run; only the **Save Clustering Parameters** task is rerun, which is very fast.

3.6.4.2 Modifying the FPGA filling rate parameters

If the compilation fails for very long zCore-level compilations or for FPGA resource issues, it may be of interest to switch the **Clustering Control** to **Automatic with Parameters** in order to adjust the parameters of the algorithm, in particular the maximum authorized filling rates for registers, LUTs and BRAMs and the ratio between Multipliers and DSP:



The image shows a software interface titled "default Back-End: Clustering". It contains several sections:

- Clustering Control:** Three radio buttons: "Pre-existing Mapping", "Automatic with Parameters" (selected), and "Full Automatic".
- Auto Core Generation:** A checkbox that is unchecked. Below it is a text field for "Advanced Auto Core Generation File" and three buttons: "...", "Use Absolute Path", "Use Relative Path", and "Edit ...".
- Algorithm:** A section with a checked checkbox. It contains four rows of sliders:
 - Registers Maximum Fill Rate (%): slider from 10 to 200, value 55.
 - LUTs Maximum Fill Rate (%): slider from 10 to 200, value 70.
 - BRAMs Maximum Fill Rate (%): slider from 10 to 200, value 80.
 - Multipliers DSP (%): slider from 10 to 200, value 80.
- Mapping Output File:** A section with a checked checkbox. It contains a text field with the path "/home/user/my_project/src/zebu/saved-mapping.xcui" and three buttons: "...", "Use Absolute Path", and "Use Relative Path".
- At the bottom are two buttons: "Advanced Settings" and "Default Settings".

Figure 42: Clustering panel for Automatic with Parameters mode

The filling rates values set in the above panel are applicable for all the automatic algorithms of partitioning: automatic zCore generation, performance oriented partitioning and density oriented clustering.

3.6.4.3 Skipping the Automatic Process

Once the compilation has completed successfully, user should skip the automatic clustering process for future compilations by switching the **Clustering Control** to **Pre-existing Mapping** if the design is not largely modified.

The **Mapping Input File** checkbox has to be selected and the path to an existing mapping file has to be added in the corresponding file input field:

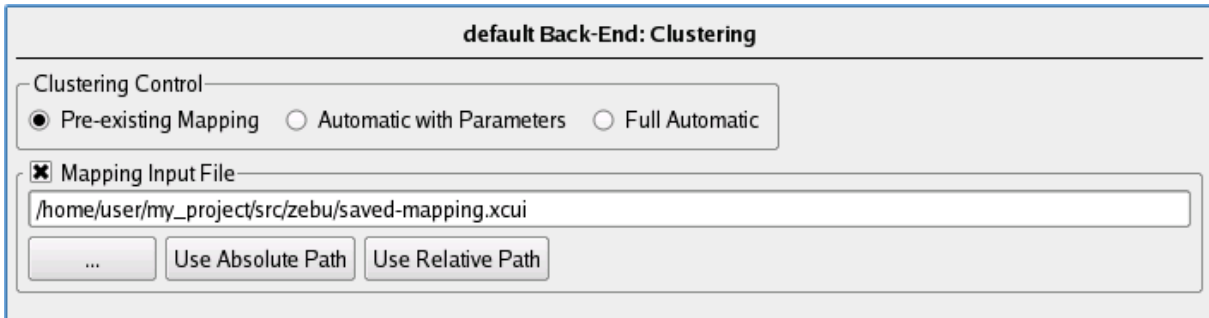


Figure 43: Clustering panel for Pre-existing Mapping mode

Since the content of the zCores may impact the compilation efficiency with a pre-existing mapping, it is recommended to switch back to one of the automatic clustering processes (automatic zCore generation and/or Density Oriented Clustering or Performance Oriented Partitioning) in case of major modifications in the source files of the design.

3.6.4.4 Modifying the Mapping File

Modifying the mapping file generated by **zCui** (.xcui file in previous sections) is possible, in particular for performance optimization purposes. However, such modifications must be done with much care because they may impact the compilation capability and/or the runtime operability of the design. Recommendations for manual optimization of the clustering solution will be available in the future.



3.7 Specific Design Modeling for ZeBu

3.7.1 Introduction

When preparing the compilation project, some specific points of the design modeling have to be checked in order to match the requirements to map the design onto the ZeBu system.

The following table shows in which **zCui** panel and/or frame and in which file the commands are declared (when applicable) for the features described in this section:

Feature	zCui panel and/or frame	Command file (if applicable)	See Section
Default registers initialization	Advanced → Build System Parameters		3.7.3.1
Individual registers initialization	zCore Definition	Netlist Edition File	3.7.3.1
Reset			3.7.3.2
Tristates (*)	Advanced → Build System Parameters	Additional Command File	3.7.4
Force (*)	zCore Definition	Netlist Edition File	3.7.5
Dynamic Force (*)	Advanced → Build System Parameters	Additional Command File	3.7.6
Design Blackbox	zCore Definition	Netlist Edition File	3.7.7
(*) When synthesizing with zFAST , the corresponding commands can be declared using RTL paths in the RTL-based compilation scripts, as described in Section 3.7.2.			

There is only one **Netlist Edition File** declared in the compilation project, which makes it necessary to create one file which gathers the commands declared for the various features. Note that the **Netlist Edition File** is interpreted sequentially by the ZeBu compiler: the order of the commands for a given feature is important.

There is also only one **Build System Parameters → Additional Command File** declared in the **Advanced** panel, which makes it necessary to create one file which gathers the commands declared for the various features.

In case some commands which should be in the **Netlist Edition File** are included by mistake in the **Additional Command File**, they will be executed correctly and a warning message will be issued. Ideally, user should move these commands to the appropriate file for future compilations.

If a command could not be executed correctly because it was in the wrong file, an error messages is displayed instead of a warning.

3.7.2 RTL-based Compilation Scripts

The **Design** → **Sources** → **RTL-based Compilation Scripts** item of the project tree can be used to declare Tcl scripts (*.tcl extension only) with the following commands:

- probe_signals with any synthesizer
- force, force_dyn and tristate when synthesizing with **zFAST**

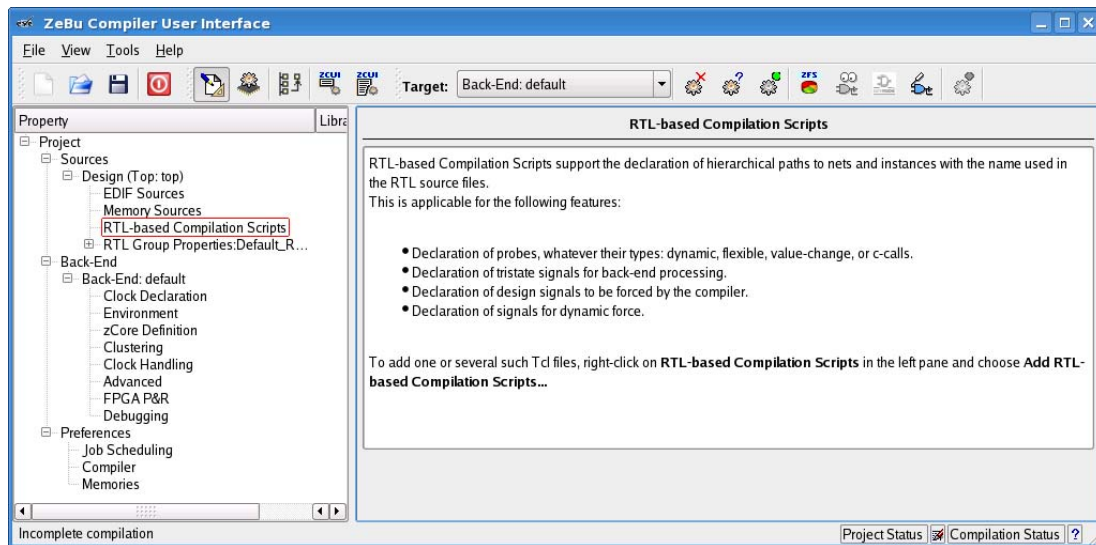


Figure 44: RTL-based Compilation Script item in Project view

Declarations based on RTL paths are applicable with specific options in the system-level compiler commands:

Commands	Options for EDIF paths	Options for RTL paths
probe_signals (applicable for all types: Dynamic, Flex, C-calls and Value Change)	-wire <signal> -wire_file <file> -port <port> -port_file <file> -instance <instance> -instance <inst> -wire <s> -instance_file <file>	-rtlname <rtl_signal> -rtlname_file <rtl_file> -rtlname <rtl_signal> -rtlname_file <rtl_file> -rtlname <rtl_instance> -instance <rtl_i> -rtlname <rtl_s> -rtlname_file <rtl_file>
probe_signals (Flex type only)	-enable_wire <signal> -enable_port <port>	-enable_rtl <rtl_signal> -enable_rtl <rtl_port>
force	-pin <port> -net <net> -pin_input <port> -pin_output <port> -source_pin <port> -source_net <net>	-rtlname <rtl_port> -rtlname <rtl_net> -rtlname_source <rtl_port> -rtlname_source <rtl_net>
force_dyn	-wire <signal> -wire_file <file> -pin <port> -pin_file <file>	-rtlname <rtl_signal> -rtlname_file <rtl_file> -rtlname <rtl_port> -rtlname_file <rtl_file>
tristate	<signal>	-rtlname <rtl_signal>

Note that options not listed in this table are not impacted by the declaration of RTL paths.

RTL paths are also supported for pattern-matching declarations (-fnmatch option), except in case of conflict between special characters in the RTL paths and pattern matching characters.



Examples:

Declaring flexible probes with an RTL path for the interface of top.instance:

```
probe_signals -type flexible -rtlname top.instance
```

Declaring flexible probes with an RTL path for a foo signal in top.instance:

```
probe_signals -type flexible -instance top.instance -rtlname {foo}
```

Declaring flexible probes with a pattern-matching RTL path:

```
probe_signals -type flexible \  
-rtlname {top.instance.in* top.instance.out*} -fnmatch
```

Declaring flexible probe with an RTL path for the signal and the enable:

```
probe_signals -type flexible -rtlname {top.instance} \  
-enable_rtl top.instance.probe_en
```

Notes:

- An RTL-based script supports both RTL paths (declared with `-rtlname` options) as well as EDIF paths (declared with the former syntax) but mixing the two may be difficult to read.
- All other compilation commands, in particular mapping constraints and connection to the Direct ICE interface, only support signals declared with EDIF paths.

3.7.3 Handling Initializations

The initialization strategy in a simulator may be slightly different from the ZeBu environment.

3.7.3.1 Registers Initialization

In ZeBu, the registers of the design are not initialized to 0 by default: they are set according to the Xilinx rules. However, it is possible to force an initialization value either manually for instances or automatically for all the non-initialized registers of the design.

To set an initialization value for register instances, the following command should be added in the **zCore Definition** → **Netlist Edition File**. This declaration can be done instance-by-instance or on a pattern-matching basis:

```
reg_init [-inst_path <my_instance>] [-value {0|1}]  
reg_init [-fnmatch] [-inst_path <match_pattern>] [-value {0|1}]
```

Where `<my_instance>` is the exact path to the instance; `-fnmatch` option is intended for pattern-matching declaration (`<match_pattern>` includes `*` for several characters and/or `?` for a single character).

Note that the hierarchical names used for this command must match the EDIF signal names. If the **Optimization vs Debugging** option for synthesis is set for maximum optimization (see Section 3.2.4.1 and 3.2.4.2), some RTL signal names may not be available in the generated EDIF netlists.

If the instance is not found or cannot be initialized, an error message is displayed in the log of the **Build System** compilation step.

Any existing initial value which may result from synthesis is overwritten by this command.

The registers that are not declared manually for initialization can be initialized by default in the **Advanced** panel:

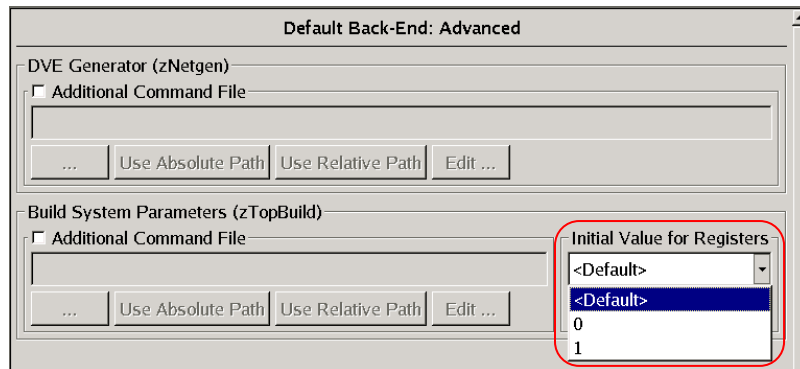


Figure 45: Initial Value for Registers in Advanced Panel

3.7.3.2 Handling the design reset

The design reset is automatically handled by the ZeBu Reconfigurable Testbench (RTB) and does not require any specific options when compiling the design.

3.7.4 Handling tristates

The ZeBu compiler guarantees that all the tristate signals at the top level or in the design are correctly driven, in particular the tristate signals with several drivers active at the same time.

If a tristate signal is in a non-deterministic state because of several drivers, the ZeBu compiler chooses the most appropriate conflict resolution: all the tristate signals of the design are processed as pull-ups and multi-driven tristate signals are processed with the wand type (if any of the drivers' values is 0, the tristate signal is driven to 0).

It is possible for the user to decide which resolution matches the expected runtime behavior of the design:

- For each tristate signal of the design, the resolution can be set to pull-down, pull-up or keeper (keeps the latest value of the signal).
- If any of the drivers' values is 0, the multi-driven tristate signal can be driven to 0: wand type.
- If any of the drivers' values is 1, the multi-driven tristate signal is driven to 1: wor type.

Conditional pull-up/pull-down and runtime-programmable pull-up/pull-down are not available in the present software version.

When the default settings do not match the design constraints, specific settings for tristate signals can be declared with the `tristate` command in the **Build System Parameters** → **Additional Command File** of the **Advanced** panel.



This declaration can be done signal-by-signal and or on a pattern-matching basis. All the constraints for a signal or for a group of signals are declared in a single command: there cannot be several `tristate` commands for the same signal or for the same group of signals.

A tristate signal can be declared individually and as a matching signal in a pattern-based declaration but different settings for the same signal are not accepted and the ZeBu compiler fails with an error message.

Note that the declaration of tristate signals uses the signal names in the EDIF netlist, which may be different from the original RTL names of the signals because of the possible optimizations during the synthesis process.

3.7.4.1 Configuring the default settings for tristate signals

When the default settings for tristate signals do not match the design constraints, these default settings can be modified to new default settings which are applicable for all tristate signals in the design (subsequent commands for individual modification of tristate signals overwrite the new default settings):

```
| tristate_default [-resolution <def_resolution>] [-conflict <def_conflict>]
```

Where `<def_resolution>` can be `pullup`, `pulldown` or `keeper` and `<def_conflict>` can be `wand` or `wor`.

If the `-conflict` option is not declared, the ZeBu compiler chooses the appropriate value to minimize the logic used in FPGAs for the actual resolution.

Example:

To connect all the tristate signals of the design as pull-downs and use the best option for multi-driven signals, use the following command:

```
| tristate_default -resolution pulldown
```

To connect all the tristate signals of the design as pull-ups and use the `wor` option for multi-driven signals, use the following command:

```
| tristate_default -resolution pullup -conflict wor
```

3.7.4.2 Signal-by-signal declaration

You can change the resolution and/or option for multi-driven signals on a signal-by-signal basis:

```
| tristate <signal_name> [-<resolution>] [-conflict <wor|wand>]
```

Where `<resolution>` is `pullup` or `pulldown` or `keeper` and `<signal_name>` is the hierarchical name of the signal in the EDIF netlist.

If `-<resolution>` or `-conflict` is omitted in the command, the `tristate_default` setting is considered for this option.

Only one tristate command is supported for a given signal in the command file, thus resolution and conflict options must be declared together if they are both necessary.

Example:

```
| tristate top.ins1.ins2.bus -pulldown -conflict wor
```




3.7.4.3 Pattern-matching declaration

You can change the resolution and/or option for multi-driven signals on a pattern-matching basis with `-fnmatch` parameter in the command line:

```
tristate -fnmatch <match_pattern> [-exclude <excl_pattern>]  
[-<resolution>] [-conflict <wor|wand>]
```

Where `<match_pattern>` is a string with `*` (for several characters) and/or `?` (for a single character); `<excl_pattern>` is an optional string with `*` and/or `?` which excludes some patterns from the declaration; `<resolution>` is `pullup` or `pulldown` or `keeper`.

Only one `tristate` command is supported for a given pattern in the command file, thus resolution and conflict options must be declared together if both are necessary.

Examples:

To define a pull-up resolution for tristate signals whose hierarchical names match `top.sys*.ins.sig*`:

```
tristate -fnmatch top.sys*.ins.sig* -pullup
```

To define resolution a pull-down resolution for tristate signals whose hierarchical names match `top.ins*.busz*`, excluding `busz1`:

```
tristate -fnmatch top.ins*.busz* -exclude busz1 -pulldown
```



3.7.5 Forcing Nets in the Design

When compiling for ZeBu, you can force any net of the design to a fixed value (0 or 1) or to a runtime-programmable value. You can also force the interconnection between nets of the design.

When forcing a net, the netlist of the design is modified by the compiler: the driver in the original netlist is automatically disconnected except if this net and its new driver are tristate signals (disconnection is optional for tristate signals).

The declaration can be done net-by-net, on a pattern-matching basis with `-fnmatch` option or using a Tcl file.

The dedicated command is `force` which can be added in the **RTL-based Compilation script** with RTL paths or in the **Netlist Edition File (zCore Definition panel)** in case of EDIF paths.

When using EDIF paths, the **Optimization vs Debugging** option for synthesis should not be set for maximum optimization (see Section 3.2.4.1 and 3.2.4.2) in order to have all the signals available in the generated EDIF netlists.

<code>force</code>	<code>-pin <port></code> <code>-net <net></code> <code>-pin_input <port></code> <code>-pin_output <port></code> <code>-source_pin <port></code> <code>-source_net <net></code>	<code>-rtlname <rtl_port></code> <code>-rtlname <rtl_net></code> <code>-rtlname_source <rtl_port></code> <code>-rtlname_source <rtl_net></code>
--------------------	---	--

Note:

When this feature is used with runtime programming capability (`-REG` option), it is based on register writing capability and has the following limitation:

- For most FPGAs in the system, it is supported when the **Enable BRAM Read&Write / Write Register / Save&Restore** item in the **Debugging** tab is enabled (it is disabled by default).
- However, this feature is never supported in FPGAs connected directly on RLDRAM in 4C and 8C/ICE modules, when RLDRAM is used by the design. No message is displayed during compilation; the write access limitation only appears during emulation runtime.

3.7.5.1 Handling the Scan Chain

An SOC most often includes dedicated logic for embedded testing through a scan chain. Such testing logic is not likely to be used when emulating the design with ZeBu but it may have an impact on the compilation process and on the emulation:

- The filling rate of the design FPGAs is increased by the testing logic.
- The additional logic for the scan chain makes the clock cones of the design more complicated and the performance of the clock handling algorithms may be strongly impacted.
- The achievable runtime frequency computed by the static timing analysis is not optimal because some false paths of the scan chain may be considered.



- The actual achievable runtime frequency may not be optimal because the scan chain logic makes clock paths longer, thus impacting the actual clock skew for clocks and for data signals.

To minimize the impact of this testing logic, it is recommended to proceed as follows:

1. In the RTL source files, the clocks of the scan chain logic should be connected to 0 so that the synthesizer removes it as early as possible in the compilation flow.
2. If the RTL source files of the scan chain logic are not available, forcing the corresponding clocks to 0 can be done during the back-end compilation, as described in Section 3.7.5.5 and 3.7.5.6.

3.7.5.2 Logging the processed nets and ports

By default, the nets and ports modified by any `force` command are not listed in the **Build System** log file.

The following command modifies the reported information for the future `force` commands (the commands before this line will not be reported):

```
force default -globalVerbose yes
```

To overwrite this default setting, the `-verbose` option can be added in the `force` command itself.

Example:

In a single command file, you can declare several `force` commands with verbose active and then return to the default setting:

```
force default -globalVerbose yes  
<list of force commands with info reported in the log file>  
[...]  
force default -globalVerbose no  
<list of force commands without info reported in the log file or  
with local -verbose option>
```

3.7.5.3 Forcing all the undriven nets and ports of the design

All the unconnected nets and ports of the design can be forced at once by adding the `force undriven` command in the **Netlist Edition File (zCore Definition panel)**. The unconnected nets will be driven either to 0, 1 or to a runtime-programmable value.

```
force undriven -value <value> [-reg_init 0|1] [-verbose]
```

Where `<value>` is 0, 1 or REG; `-verbose` option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **Build System** log file; `-disconnect` option applies only when forcing tristate signals.

When forcing to a runtime-programmable value (`-value REG`), an optional initialization value can be added with the `-reg_init` option.



3.7.5.4 Getting Information about undriven nets of the design

To check which are the undriven nets in the design or to process them manually with force commands, the **Build System** step can generate a Tcl file in which all the unconnected ports and nets of the design are listed.

To get a file for the undriven ports of the design:

```
query_undriven -pin -file <path_to_file>
```

To get a file for the undriven nets of the design:

```
query_undriven -nets -file <path_to_file>
```

To get a file for undriven ports and nets of the design:

```
query_undriven -pin -nets -file <path_to_file>
```

As for any other path declared in a command file, only absolute paths should be used for <path_to_file> in the query_undriven command.

3.7.5.5 Assigning a fixed value to nets

You can force any net of the design to a fixed value (0 or 1) or to a runtime-programmable value.

For a net-by-net declaration, either with an RTL path or an EDIF path:

```
force assign -rtlname <signal> -value <value> [-verbose]
[-reg_init 0|1] [-disconnect]
force assign -net <signal> -value <value> [-verbose]
[-reg_init 0|1] [-disconnect]
```

For a pattern-matching declaration, either with an RTL path or an EDIF path:

```
force assign -fnmatch -rtlname <signal_pattern> -value <value>
[-reg_init 0|1] [-verbose] [-disconnect]
force assign -fnmatch -net <signal_pattern> -value <value>
[-reg_init 0|1] [-verbose] [-disconnect]
```

Where <value> is 0, 1 or REG; -verbose option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **System Build** log file; <signal_pattern> includes * for several characters and/or ? for a single character; -disconnect option applies only when forcing tristate signals.

When forcing to a runtime-programmable value (-value REG), an optional initialization value can be added with the -reg_init option.

3.7.5.6 Assigning a fixed value by listing nets in a separate file

```
force assign -file_name <signal_file> -value <value>
[-reg_init 0|1] [-verbose] [-disconnect]
```

Where <value> is 0, 1 or REG; -verbose option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **System Build** log file; -disconnect option applies only when forcing tristate signals.

When forcing to a runtime-programmable value (-value REG), an optional initialization value can be added with the -reg_init option.



In the file which lists the signals (<signal_file>), each line has the following format:

```
-net <signal> -value <value> [-verbose]  
[-reg_init 0|1] [-disconnect]
```

Where <value> is 0, 1 or REG; -verbose option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **Build System** log file; -disconnect option applies only when forcing tristate signals.

When forcing to a runtime-programmable value (-value REG), an optional initialization value can be added with the -reg_init option.

3.7.5.7 Assigning a testbench value to a net

In order to drive a net in the design from the testbench, it is possible to force it with an interface port declared in the DVE file. The net can be declared either explicitly with its hierarchical path (-net option) or using the name of a port in the design connected to the net (-pin option):

```
force assign -source_dve <src> -net <signal_load> [-disconnect] [-  
verbose]  
force assign -source_dve <src> -pin <port_load> [-disconnect] [-  
verbose]
```

Where <src> is the name of a port declared in the DVE file; <signal_load> is the exact hierarchical name for the forced net; <port_load> is the hierarchical name of the port; -verbose option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **System Build** log file; -disconnect option applies only when forcing tristate signals.

It is not possible to use pattern-matching declaration with the -source_dve option.

3.7.5.8 Interconnecting Nets

You can interconnect two nets of your design, assigning a source net to a load net:

```
force assign -source_net <src> -net <load> [-disconnect] [-verbose]
```

Where <src> and <load> are the exact hierarchical names for the interconnected nets; -verbose option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **Build System** log file; -disconnect option applies only when forcing tristate signals.

Pattern-matching declaration is also supported for the source and load nets with -fnmatch option, with the condition that there must be as many sources as nets to be forced, or only one source for all the nets to be forced:

```
force assign -fnmatch -source_net <src_pattern> -net <load_pattern>  
[-disconnect] [-verbose]
```

Where <src_pattern> and <load_pattern> are strings with * (for several characters) and/or ? (for a single character); -verbose option can be added to modify the default setting to have the signals and ports processed by the commands listed in the **System Build** log file.



3.7.6 Declaring Signals for Dynamic Control at Runtime (Dynamic Force)

When bringing up the design, it may be of interest to control at runtime if a signal is driven as defined in the source files or to force this signal from the testbench. Once declared during compilation, this control capability can be changed dynamically during emulation runtime. This feature is also known as “Dynamic Force”.

This control is based on the declaration of nets or ports of instances in the design and supports pattern-matching declaration. The dedicated command is `force_dyn` which can be added in the RTL-based Compilation script with RTL paths or in the **Build System Parameters** → **Additional Command File (Advanced)** panel in case of EDIF paths.

When using EDIF paths, the **Optimization vs Debugging** option for synthesis should not be set for maximum optimization (see Section 3.2.4.1 and 3.2.4.2) in order to have all the signals available in the generated EDIF netlists.

By default, any “point” connected to the declared net is impacted when the control signals are modified at runtime, as described in Section 3.7.6.5. This way of applying the dynamic force is known as “global”. In such a case, the runtime database provides access to each of the points with its local name and the values are all forced at once.

When this command is processed by the system-level compiler, the report file includes a dedicated section which is named “User dynamic forces” (Section 5.1 of the report). This is of major interest in case of pattern-matching declarations.

Example:

```
5.1.User dynamic forces.
=====
2 dynamic force command(s) were specified by user. Only processing on non-quiet commands will
be displayed.
Preprocessing command force_dyn -wire "dut.en_r" -type global
=> 1 dynamic forces will be inserted.
Preprocessing command force_dyn -wire "dut.etage_1.en_r" -type global
=> 1 dynamic forces will be inserted.
Processings from command 'force_dyn -wire "dut.en_r" -type global'.
Instrumenting wire 'dut.en_r' in global mode.
    Disconnecting driver dut.en_r_reg.Q
Processings from command 'force_dyn -wire "dut.etage_1.en_r" -type global'.
Instrumenting wire 'dut.etage_1.en_r' in global mode.
    Disconnecting driver dut.etage_1.en_r_reg.Q
2 dynamic forces were inserted.
```

Note:

This feature is based on register writing capability and has the following limitation:

- For most FPGAs in the system, it is supported when the **Enable BRAM Read&Write / Write Register / Save&Restore** item in the **Debugging** tab is enabled (it is disabled by default).
- However, this feature is never supported in FPGAs connected directly on RLDRAM in 4C and 8C/ICE modules, when RLDRAM is used by the design. No message is displayed during compilation; the write access limitation only appears during emulation runtime.



3.7.6.1 Declaring Dynamic Force for Nets

To provide control capability at runtime for nets, the `force_dyn` command can be declared with the hierarchical path to the signal:

```
| force_dyn -wire <path_to_net>
```

Where `<path_to_net>` is the exact hierarchical path to the net.

For pattern-matching declaration, the `-fnmatch` option should be used:

```
| force_dyn -fnmatch -wire <net_pattern>
```

Where `<net_pattern>` is a string with `*` (for several characters) and/or `?` (for a single character); note that pattern-matching is done for each hierarchical level and does not include the separator.

Note that `-net` option is equivalent to `-wire` and can be used in a similar way.

3.7.6.2 Declaring Dynamic Force for Ports

To provide control capability at runtime for the nets connected to a port of an instance, the `force_dyn` can be declared with the hierarchical path to the port:

```
| force_dyn -pin <path_to_port>
```

Where `<path_to_port>` is the exact hierarchical path to the port.

For pattern-matching declaration, the `-fnmatch` option should be used:

```
| force_dyn -fnmatch -pin <port_pattern>
```

Where `<port_pattern>` is a string with `*` (for several characters) and/or `?` (for a single character); note that pattern-matching is done for each hierarchical level and does not include the separator.

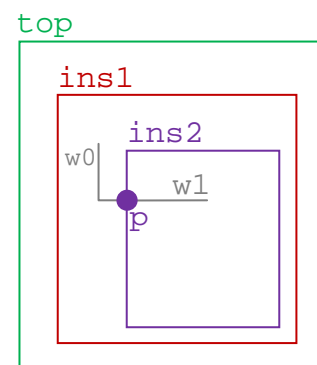
Note that `-port` option is equivalent to `-pin` and can be used in a similar way.

The net which is actually modified for dynamic forcing is the one which is at the same hierarchical level as the instance.

Example:

```
| force_dyn -pin top.ins1.ins2.p
```

The net that will be actually available for runtime control is `w0`.





3.7.6.3 Downstream control

When the global control of a net does not match the specific environment constraints, it is possible to control only the load signals of the forced net.

For that purpose, the `-downstream` option should be added to the command:

```
force_dyn -wire <path_to_net> -type downstream  
force_dyn -fnmatch -wire <net_pattern> -type downstream
```

In such a case, the ZeBu compiler automatically searches for the drivers and loads of the net in order to control only the loads which are in the hierarchical level “below” `<path_to_net>` and not in other hierarchical levels in the design.

Example:

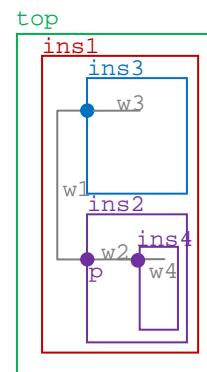
```
force_dyn -wire top.ins1.ins2.w2
```

w2 is available for runtime control and by default w1, w3 and w4 are also set to the same value.

```
force_dyn -wire top.ins1.ins2.w2 -type  
downstream
```

If the actual driver is on w1, w2 is available for runtime control and w4 is set to the same value; w1 and w3 are driven by the design, in spite of the dynamic force.

If the actual driver is on w2 is the actual driver, w2 is available for runtime control and w1, w3 and w4 are set to the same value.



The criteria to search for drivers and loads are based on the connection of the net to FPGA gates in the netlist. If the net has multiple drivers or in case of equipotential loops, the ZeBu compiler stops with an error.

3.7.6.4 Adding a dynamic probe on a signal declared for runtime control

In order to monitor the signal, in particular to know its level when driven by the design, it is possible to add the `-probe_forced` option in the command line:

```
force_dyn -wire <path_to_net> -probe_forced
```

Where `<path_to_net>` is the exact hierarchical path to the net.

3.7.6.5 Runtime control

The ZeBu compiler adds the appropriate elements in the runtime database to control check and modify the driver of the signal and the user-defined value when the signal is not driven by the design.

At runtime, these elements can be accessed as dynamic probes from `zDbPostProc` or from the testbench to control:

- `DynForce.<path_to_signal>.dyn_force_enable:`
When set to 0, the signal is driven by the design.
When set to 1, the signal is forced to `DynForce.<path_to_signal>.dyn_force_value`
- `DynForce.<path_to_signal>.dyn_force_value:` user-defined value for the signal when not driven by the design.



The C++ and C runtime API also provides the corresponding methods and functions to control the signals declared with `force_dyn` command from the testbench: check if a signal can be forced or not, enable/disable manual force, set value for manual force.

The corresponding C++ methods are part of the `Signal` class, and the C functions are prefixed with `ZEBU_Signal`. They are included with `libZebu.hh/libZebu.h`. They are described in the [ZeBu C++ API Reference Manual](#) and [Zebu C API Reference Manual](#).

3.7.7 Processing Design Blackboxes

The EDIF netlist resulting from synthesis may instantiate some empty modules known as “blackboxes”. In most cases, blackboxes exist for the following types of modules:

- Non-synthesizable modules such as PLLs
- IP modules provided as Xilinx pre-compiled netlists (ngo files).
- Modules for which the RTL source files are not yet available.

By default, the ZeBu compiler generates an error and stops when a blackbox is present in the EDIF netlist. It is possible to declare how a blackbox module will be processed by the Zebu compiler:

- The blackbox module can be kept as-is through the back-end compilation and replaced by its content during FPGA Place & Route.
- The connection of the blackbox module to the design is declared by user so that new drivers are inserted in replacement of the outputs of the blackbox module. The new drivers can be constants, runtime-programmable registers declared in the DVE or runtime driven by the testbench or by HDL simulator. The existing connections to inputs of the blackbox module are not modified.

The declaration of blackbox modules is done in the **zCore Definition → Netlist Edition File** with the `blackbox` command. The order in which the commands are processed by the ZeBu compiler is the same as the order in which they are declared in this file. The declaration is done module-by-module or on a pattern-matching basis with `-fnmatch` option.

The following rules apply when processing EDIF blackboxes:

- If blackboxes are detected but not declared explicitly or implicitly, an error is generated and the ZeBu compiler stops.
- If modules not instantiated in the design are declared as blackboxes, a warning is generated (with `-verbose` option only).
- If modules containing design information are declared as blackboxes, an error is generated and the ZeBu compiler stops.

To have detailed information about blackbox processing in the **Build System** log file, the `-verbose` option must be added to each `blackbox` command.



Note:

When this feature is used with runtime programming capability (-REG option), it is based on register writing capability and has the following limitation:

- For most FPGAs in the system, it is supported when the **Enable BRAM Read&Write / Write Register / Save&Restore** item in the **Debugging** tab is enabled (it is disabled by default).
- However, this feature is never supported in FPGAs connected directly on RLDRAM in 4C and 8C/ICE modules, when RLDRAM is used by the design. No message is displayed during compilation; the write access limitation only appears during emulation runtime.

3.7.7.1 Keeping the blackboxes through back-end compilation

An IP which is integrated as an NGO file in the design, or a module of the design which is not yet available for emulation, have to be kept as blackbox modules in the netlist until FPGA Place and Route.

The blackbox module has to be declared explicitly and an additional command can be added to be sure that it will not be processed in a later pattern-based declaration in the file. The syntax of the commands to be added in the **zCore Definition** → **Netlist Edition File** is the following one:

```
blackbox define <module_name>  
blackbox ignore <module_name>
```

For a pattern-matching declaration, the syntax is the following one:

```
blackbox define -fnmatch <module_pattern>  
blackbox ignore -fnmatch <module_pattern>
```

Where <module_pattern> includes * for several characters and/or ? for a single character.

It is not recommended to declare only * for <module_pattern> because all empty modules would be detected as blackboxes in the design and will be kept through the back-end compilation. The blackbox processing during back-end compilation would not detect the empty modules (if any) and any erroneous empty module would be detected only during FGPA Place and Route.

3.7.7.2 Removing Blackboxes

A blackbox module can be removed by the ZeBu compiler, and its output and inout ports optionally forced:

```
blackbox remove <module_name> [-value <port_value>]  
blackbox remove -fnmatch <module_pattern> [-value <port_value>]
```

Where <module_name> is a string; <module_pattern> includes * for several characters and/or ? for a single character; <port_value> can be 0, 1, REG (runtime-programmable value), Z (high-impedance with runtime control) or NONE (leaves the ports unconnected).

When the command is set to have a runtime-programmable value (-value REG option) or high impedance (-value Z option), the runtime database includes the



appropriate elements for runtime access (from **zRun** or from the testbench) for each output or inout port of the blackbox module:

- -value REG option: <path_to_blackbox>.<bb_portname>.zBB_value
- -value Z option: <path_to_blackbox>.<bb_portname>.zBB_enable and <path_to_blackbox>.<bb_portname>.zBB_value

Using * for <module_pattern> means that all blackboxes detected in the design will be removed, with their output and inout ports driven with the same -value option.

3.7.7.3 Disconnecting a Port from a Blackbox

When some multi-driver errors are reported for a blackbox in the design, an output or inout port can be disconnected from all instantiations of the blackbox module, and the corresponding nets forced to 0 or 1. In such a case, the instances of the blackbox still appear in the design.

The declaration of ports can be done port-by-port or on a pattern-matching basis (with -fnmatch option).

```
blackbox drive <module_name> [-port {<list_of_ports>}] [-value  
<port_value>]
```

```
blackbox drive -fnmatch <module_pattern> [-port  
{<list_of_ports_with_pattern>}] [-value <port_value>]
```

Where <port_value> can be 0, 1, REG (runtime-programmable value), Z (high-impedance with runtime control) or NONE (leaves the ports unconnected).

If the -port option is not present, all the output and inout ports of the declared module(s) are disconnected. If an input port is declared for the -port option, the ZeBu compiler ignores it.



3.8 Memory Modeling

3.8.1 Introduction

In ZeBu, design and testbench memories can be implemented in different ways depending on size and performance criteria:

- Small size memories (from a few bits to 1 kBytes) based on registers or LUTRAM (also known as Distributed RAM) in the Xilinx Virtex FPGAs.
- Medium size memories (from 1 to 500 kBytes) based on BlockRAM (BRAM) in the Xilinx Virtex FPGAs.
- Large size memories (exceeding 500 kBytes) based on on-board memory resources (called zrm memories), using DDR2 memory banks and fast RLDRAM memory chips.

All these memory implementations offer a rich set of SRAM-type memory models which are generated by the memory generator of the ZeBu Compiler.

Additional ZeBu memory models for complex memories (for example DDRx/GDDRx SDRAM and NAND/NOR Flash) are available as separate IPs; specific documentation is available from your usual EVE representative.

If you need an ultra-large memory which exceeds the memory capacity of ZeBu-Server, typically for processor or printer applications, you can take advantage of the memory resources of the host PC with a dedicated transactor. The modeling of such a memory with a memory transactor is beyond the scope of this manual. Refer to the dedicated manual which is available from your usual EVE representative.

3.8.2 Implementation Guidelines

In most cases, the design memories are implemented automatically by the ZeBu compiler (in particular when compiling with **zFAST**).

When choosing a memory implementation in the ZeBu environment, it is important to consider the required memory size for verification, which may be different from the memory size targeted in the final design.

The choice between LUTRAM-based and BRAM-based memory models is done automatically by the ZeBu compiler on size criteria. This selection is fully automatic when synthesizing with **zFAST**. It can be selected by user when creating a **Memory Source** file for the ZeBu memory generator, as described in Section 3.8.5

The choice between BRAM-based and zrm-based memory models is mostly based on size criteria but performance criteria can be taken into account as well. For example, a 1-MByte memory would use 228 of the 286 BRAM memory resources available in one FPGA. However, a BRAM-based memory always provides a higher runtime performance than a zrm memory.

The choice between DDR2 and RLDRAM memory resources is fully transparent for the user since it is processed by the automatic clustering, based on size criteria.

Table 3: Memory Implementation Guidelines

Size Range	<kB	<10kB	<500kB	>500 kB	>500 kB
Recommended Memory Resources	FPGA-based memories			zrm-based memories	
	Registers	LUTRAM	BRAMs	RLDRAM	DDR2
# of instances	200k/FPGA	100k/FPGA	288/FPGA	module dependant	module dependant
Size of instance	1 bits	64 bits	36 kbits	256/128 MBytes	4/2 GBytes
max # of ports		128 (recommended)	128 (recommended)	8	8
Type of ports	synchronous asynchronous	synchronous asynchronous	synchronous asynchronous	synchronous	synchronous
Speed	+++	+++	+++	++	+
Runtime access	Register access	✓ (if created with ZeBu memory generator)	✓ (if created with ZeBu memory generator)	✓	✓

ZeBu-Server supports multi-port memories without any constraint for the maximum number of ports for memories based on the memory resources of the Xilinx Virtex FPGAs and with up to 8 ports for zrm memories. However, it is not recommended to use memories with a very high number of ports. For example a 32-port memory will most probably impact the achievable runtime frequency and the clustering and FPGA place and route steps may fail for a 128-port memory.

Memory models based on LUTRAMs or BRAMs can be defined as synchronous and/or asynchronous. In general, synchronous models are preferred to asynchronous models since the ZeBu system has been designed for functional verification and not for timing verification and because asynchronous memory models may cause some runtime instability due to hold time/setup time issues. Moreover, using synchronous memories shortens the compilation time because no specific timing constraints are required during the FPGA Place and Route phase.

3.8.3 Automatic Memory Modeling when synthesizing with **zFAST**

When synthesizing with **zFAST**, the memory instances of the design are automatically processed in the ZeBu compilation process:

- Memories smaller than 2048 bits are automatically inferred as register-based memory models.
- For bigger memories, the ZeBu compiler uses either FPGA embedded memory resources (LUT-based memories or Block-RAM (BRAM) based memories) or the zrm memories. The most appropriate resource is automatically selected during compilation by the ZeBu memory generator.

For specific design requirements, the threshold between register-based memories and ZeBu memory models (based on LUTRAM or on BRAM memory resources) can be modified: **Resource Memory Inference** → **zMem memory** selector in **RTL Group** → **zFAST** tab.

All the design memories created when synthesizing with **zFAST** can be accessed easily at runtime and they can be initialized automatically if the RTL source files include the appropriate `readmemb/readmemh` instructions.

Information about the type of memory actually created by the ZeBu memory generator is available from the contextual menu for **Build System** step (choose **Show zTopBuild HTML report** item and go to section 3.5 – zMem resource utilization).

As stated in Section 3.8.6.3, synchronous memory models are recommended in ZeBu. However, when the RTL source files instantiate a memory with asynchronous writings, **zFAST** automatically infers asynchronous memory models and may cause some instability issues.

Note that some **zFAST** attributes can be added in the **zFAST Additional Attribute File** to control the memory inference, as described in the ***zFAST Synthesizer Manual*** (Section 6.1.9 in Rev A).

3.8.4 Inputs of the ZeBu Memory Generator

The ZeBu memory generator provides LUTRAM-based, BRAM-based or zrm-based memory models from Tcl descriptions of the memories:

- When synthesizing with **zFAST**, these memory source files are automatically generated from the RTL description of the memories.
- The Tcl file can be written manually and declared as **Memory Sources** of the project in **zCui**:

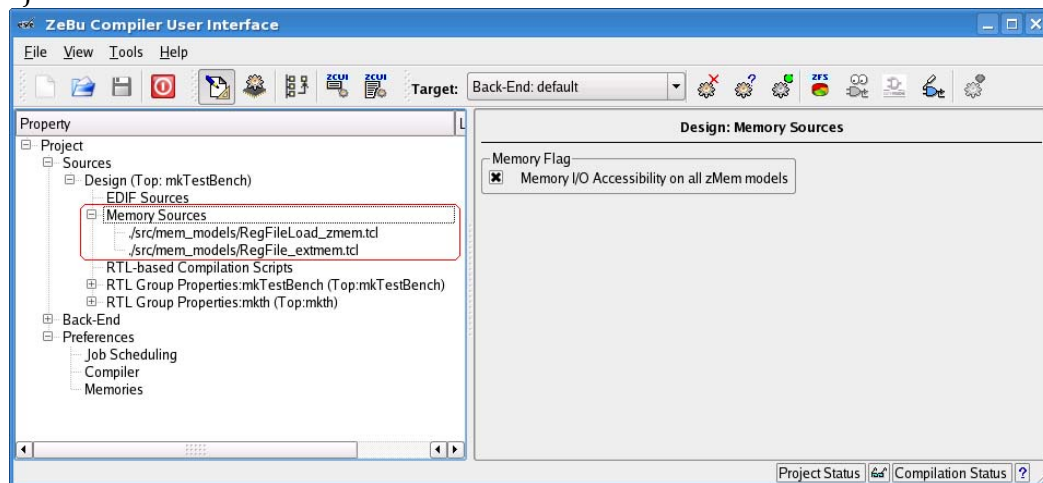


Figure 46: Memory Sources Panel

Note that the Tcl memory source files generated by **zFAST** do not appear in the **Memory Sources** in **zCui**.

By default, all the ports of the memories based on models from the ZeBu memory generator can be accessed at runtime through dynamic probes. This feature can be disabled by clearing the **Memory Flag** checkbox which is shown in Figure 46 or by adding specific commands in the memory source file, as described in Section 3.8.6.9.

3.8.5 Settings for automatic selection of the memory physical resources

When the memory are automatically inferred by **zFAST** or when the automatic mode is explicitly requested in the Tcl memory source file, the ZeBu memory generator chooses the most appropriate physical memory resources to implement the memory model (LUTRAM-based, BRAM-based or zrm-based memory model), based on size and number of ports criteria or the limit for BRAM usage set in the **Preferences** → **Memories** panel:

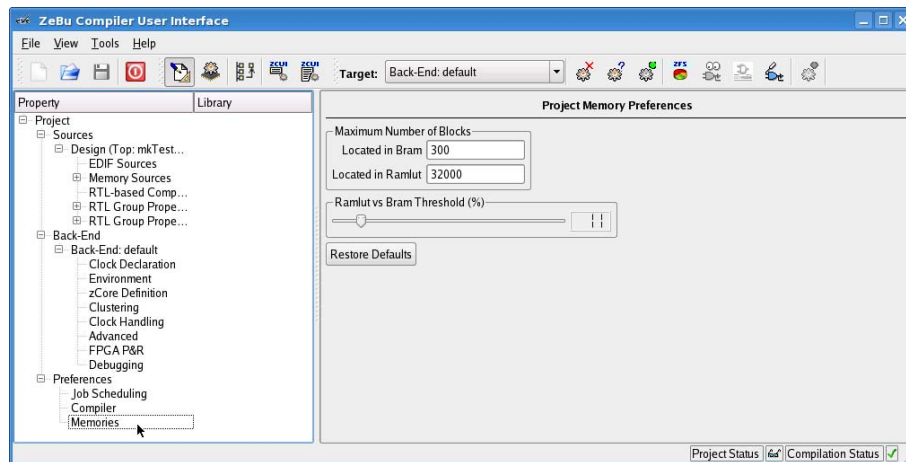


Figure 47: Preferences → Memories panel

The **Maximum Number of Blocks** frame displays the maximum authorized size (in blocks) for each BRAM-based and LUTRAM-based memory model. The default values are set so that one memory model can be mapped on a single FPGA:

- The default value for BRAM resources is 300 per FPGA.
- The default value for RAMLUT resources is 32000 per FPGA.

The **Ramlut vs Bram Threshold (%)** frame defines a preference ratio between LUTRAM and BRAM based on a usage criteria: since BRAM memories are often critical resources in the FPGAs, it is important to map memory models on them only if the physical resources are not wasted because their actual usage is low. As a consequence:

- Increasing the threshold value (default is 11%) will select preferably LUTRAM-based memories.
- Setting the threshold value at 0 will map all models with automatic selection to BRAMs.
- Setting the threshold value at 100% will map all models with automatic selection to LUTRAMs.

If the ZeBu memory generator cannot generate the expected memory model with the automatic mode default settings, an error message is displayed which recommends that the user should choose himself the type of memory in the memory new command.

3.8.6 Writing a Memory Source file

The Tcl script which described a memory model for the ZeBu memory generator can be written manually for LUTRAM-based, BRAM-based or zrm-based memory models.

The output of the ZeBu memory generator is an EDIF memory model that is automatically processed by the back-end compiler in replacement of the memory instances in the design.

A memory source file includes the following:

- Targeted memory type (LUTRAM, BRAM or zrm) or activation of the automatic selection of the type.
- Declaration of the characteristics of the memory model (depth, width, word length...) with specific capability according to the type of memory.
- Declaration of each port of the memory model with its type (read-only, write-only or read/write) and configuration (access priority, synchronous/asynchronous ...)
- For each port of the memory, declaration of the interface signals.
- Settings for optimization and accessibility.
- Command for generation.

The following sub-sections list the available commands that should be added in the Memory Source File and some complete script examples are also available in Sections 3.8.6.13 to 3.8.6.14.

Note that the Zebu memory generator can also be used as an interactive tool but this mode is not described in the present documentation package.

3.8.6.1 Selection of the type of memory

In the Memory Source file, the type of memory resource of the memory model can be declared explicitly or the automatic mode of the ZeBu memory generator can be selected. This is done through the command which creates the memory model:

```
memory new <myMemory> [<type>]
```

Where <myMemory> is the name of the memory model as it is instantiated in the design; <type> is one of the following values: bram, ramlut, zrm or auto (if <type> is not set, the memory generator switches to automatic mode).



3.8.6.2 Characteristics of the memory

The characteristics of the memory: depth (number of words), width (size of a word) and type (synchronous or asynchronous), are declared with the following commands:

```
memory depth <my_depth>
memory width <my_width>
memory type <my_type>
```

Where:

<my_depth> is the number of words. It must be a power of 2 (if not, it will be rounded up to the next power of 2).

<my_width> is the size of a word, expressed as a number of bits.

<my_type> can be declared for LUTRAM-based and BRAM-based models; possible values are sync (for a synchronous memory) or async (for an asynchronous memory); default value is sync.

You can optionally declare the width of the data sub-word accessed with the signal (default is 8, to match the usual byte size):

```
memory set_word_length <subwordlength>
```

Where:

<subwordlength> is the number of bits of the data sub-word selected with the signal (must be a divisor of <my_width>)

3.8.6.3 Creation of the ports of the memory

At least one port must be created for each memory model; the maximum number of ports varies with the memory type: up to 32 ports can be created for a BRAM-based memory model. The options for the type and access mode can be different for each port in a multi-port memory. Note that the set_... commands are optional and their applicability is different according to the type of memory model:

```
memory add_port <myPort>
memory set_port_type <myPort> <my_port_type>
memory set_port_access <myPort> <my_port_access>
memory set_rw_mode <myPort> <my_rw_mode>
memory set_latency <myPort> <my_port_latency>
```

	LUTRAM-based model	BRAM-based model
<my_port_type> default	r (read-only) w (write-only) rw (read/write) rw	r (read-only) w (write-only) rw (read/write) rw
<my_port_access> default	sync async see memory type command	sync async see memory type command
<my_rw_mode> default	readbeforewrite noreadonwrite readafterwrite readafterwrite	readbeforewrite noreadonwrite readafterwrite readafterwrite
<my_port_latency> default	Not supported -	Not supported -

	RLDRAM zrm-based model	DDR2 zrm-based model
<my_port_type>	r (read-only) w (write-only) rw (read/write)	r (read-only) w (write-only) rw (read/write)
default	rw	rw
<my_port_access>	Not supported	Not supported
default	–	–
<my_rw_mode>	readbeforewrite	readbeforewrite
default	readbeforewrite	readbeforewrite
<my_port_latency>	[1..4]	[1..4]
default	–	–

3.8.6.4 Interface of a LUTRAM- or BRAM-based memory port

The following figure shows the functional architecture of a port of a BRAM-based memory model created with the ZeBu memory generator:

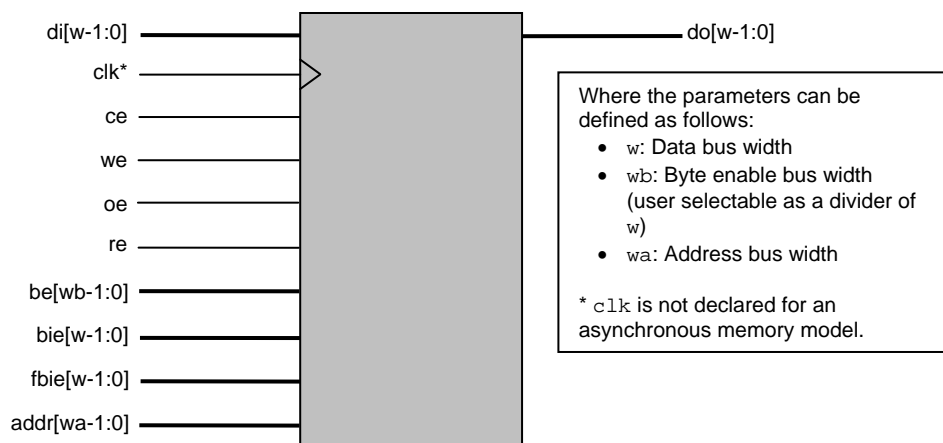


Figure 48: LUTRAM- and BRAM-based Memory Port Functional Architecture

The signals defined for a LUTRAM- or BRAM-based memory models are listed in the following table.

Table 4: Signals Description for BRAM-based Memory Models

Signal Name	Description	Port Mode			Options
		r	w	rw	
do	Data Out	x		x	
di	Data In		x	x	
addr	Address	x	x	x	
we	Write Enable *		x	x	[high low]
ce	Chip Enable *	x	x	x	[high low]
oe	Output Enable *	x		x	[high low]
re	Read Enable	x		x	[high low]
be	Byte Enable		opt	opt	[high low]
bie	Bit Enable		opt	opt	[high low]
fbie	See description below		opt	opt	[high low]
clk	Clock	x	x	x	[posedged negedge]

* When the required control signals (we, ce, oe) are not explicitly defined, they are automatically set active in the memory model.

Description of fbie:

If you intend to use only byte access for a bit-enable memory, you can lower the usage of memory resources by connecting bit-enable signals of your DUT to fbie signals of the memory model.

Note that it is not possible to declare both bie and fbie signals for the same port of a memory model.

Important Notes for Asynchronous Memories:

- There is no clock signal declaration for an asynchronous memory, but at least one write activation signal (we, be, bie or fbie) must be explicitly declared.
- The control signals must be generated in the FPGA in which the memory is mapped.

3.8.6.5 Interface of a zrm-based Memory Model

The following figure shows the functional architecture of a port of a zrm-based memory model created with the ZeBu memory generator.

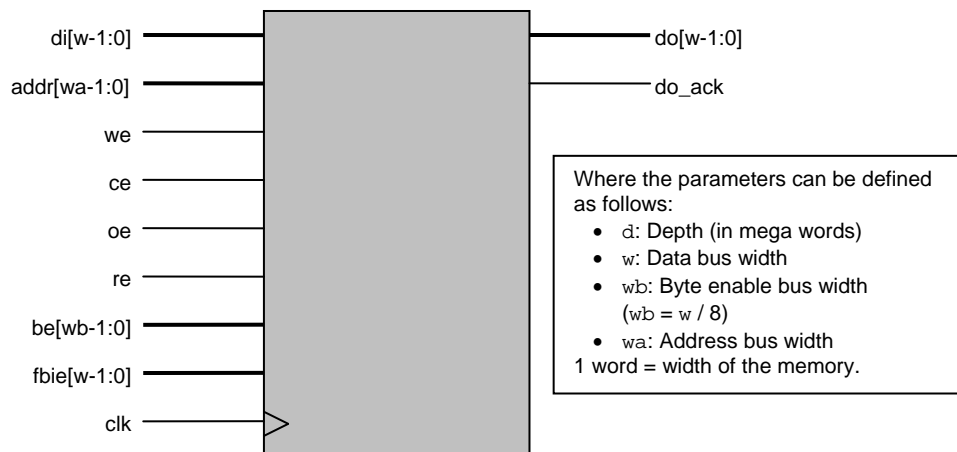


Figure 49: Memory Port Functional Architecture

The signals defined for the zrm-based memory models are listed in the following table.

Table 5: Signals Description for zrm-based Memory Models

Signal Name	Description	Port Mode			Options
		r	w	rw	
do	Data Out	x		x	
di	Data In		x	x	
addr	Address	x	x	x	
do_ack	Data Out Acknowledge	opt		opt	
we	Write Enable *		x	x	[high low]
ce	Chip Enable *	x	x	x	[high low]
oe	Output Enable *	x		x	[high low]
re	Read Enable	x		x	[high low]
be	Byte Enable		opt	opt	[high low]
fbie	See Description below		opt	opt	[high low]
clk	Clock	x	x	x	[posedged negedge]

* When the required control signals (we, ce, oe) are not explicitly defined, they are automatically set active in the memory model.



Description of fbie:

You can define a bit-enable memory model and use it for byte access only by connecting bit-enable signals of your DUT to fbie signals of the **zMem** memory model. Such memory models use less memory resources than an actual bit-enable memory model would.

3.8.6.6 Declaring the Interface Signals of a Memory Port

For each port previously defined, all the interface signals are declared with the `memory_port` command:

```
memory_port <myPort> <signal> <pin_name> [options]
```

Where <pin_name> is the user-name for <signal> at the interface of the memory model and <signal> is one of the signal names listed in Table 4 and Table 5, together with the appropriate [options].

Note: When instantiating the memory model in your DUT, connect the <pin_name> signals to the appropriate DUT signals.

3.8.6.7 Resource Optimization for BRAM-based memory models

By default, the ZeBu memory generation gives preference to performance versus logic optimization. However, it is possible to optimize the amount of memory blocks used for BRAM-based memory models in the FPGAs by adding the following command in the memory source file:

```
memory optimize_capacity true
```

3.8.6.8 Scalarizing the interface of a Memory Model

For accessibility purpose, it is possible to generate a memory model with scalar signal names instead of vectors (for example, D[7:0] will be available as D7, D6 ... D1, D0). The following command should be added in the memory source file:

```
memory scalarize true
```

Note that all the signals of the current memory are scalarized when this command is added.

3.8.6.9 Runtime Accessibility

In addition to the checkbox which is available in the **Memory Flags** frame (**Memory Source Files** panel) which is described in Section 0, the accessibility through dynamic probes can be added with more accuracy for a port or for specific signals of a memory model:

- Add visibility to the interface of the current memory:

```
memory set_memory_debug_mode true|false
```

The default value for this option is false.

- Add visibility to a specific memory port of the current memory:

```
memory set_port_debug_mode <port> true|false
```

The default value for this option is false.



- Add visibility to some specific signal of the of the current memory:

```
memory set_signal_debug_mode <signal_name> true|false
```

Where <signal_name> is the name of the signal you want to access with a dynamic probe at run-time. The default value for this option is false.

- Add visibility to all the memories of the current script :

```
memory set_global_debug_mode true|false
```

The default value for this option is false.

3.8.6.10 Modifying the interface to match the DUT requirements

When the interface of a design memory in the DUT requires some additional signals, they can be declared as extra pins for the memory model. This extra pin will not be connected inside the memory.

```
memory add_extra_pin <pin-dir> <pin-name> [left] [right]
```

Where <pin-dir> is the direction of the port (input, output, inout, vcc_output, or gnd_output). <pin-name> is the name of the port. left and right are optional parameters used to instantiate a vector (if they are not specified, a single pin will be instantiated).

3.8.6.11 Launch the generation

The command to actually launch the ZeBu memory generation is mandatory at the end of the memory source file:

```
memory generate
```

3.8.6.12 Script Example for a synchronous, multi-port, BRAM-based memory model

The following script corresponds to a synchronous, BRAM-based memory model with 3 ports (A, B and C), with the following characteristics:

- The memory size is 1024 words, with 32-bit words.
- Port A is a read-write port, with read-after-write mode; the clock for port A is clka, active on positive edge; the connected signals are addra, dia, doa, and wea is a write enable.
- Port B is a write only port; the clock is clk b, active on negative edge; the connected signal are addr b, dib; web is an active-high write enable; bie b is an active-low bit-enable.
- Port C is a read only port; the clock is clk c, active on positive edge; the connected signal are addr c, doc; oec is an active-low output enable.



```
memory new memName bram
memory depth 1024
memory width 32

memory add_port A rw
memory set_rw_mode A readafterwrite
memory_port A clk clka posedge
memory_port A addr addra
memory_port A di dia
memory_port A do doa
memory_port A we wea low

memory add_port B w
memory set_rw_mode B readafterwrite
memory_port B clk clkb negedge
memory_port B addr addrb
memory_port B di dib
memory_port B we web high
memory_port B bie bieb low

memory add_port C r
memory set_rw_mode C readafterwrite
memory_port C clk clkc posedge
memory_port C do doc
memory_port C addr addrc
memory_port C oe oec low

memory generate
```

3.8.6.13 Script Example for an asynchronous, multi-port BRAM-based memory model

The following script corresponds to an asynchronous, BRAM-based memory model with 3 ports (A, B and C), with the following characteristics:

- The memory size is 1024 words, with 32-bit words.
- Port A is a read-write port, with read-after-write mode; the connected signals are addra, dia, doa, and wea is an active-low write enable.
- Port B is a write only port; the connected signal are addrb, dib; web is an active-high write enable; bieb is an active-low bit-enable.
- Port C is a read only port; the connected signal are addrc, doc; oec is an active-low output enable.

```
memory new memName bram
memory depth 1024
memory width 32
memory type async

memory add_port A rw
memory set_rw_mode A readafterwrite
memory_port A addr addra
memory_port A di dia
memory_port A do doa
memory_port A we wea low
```



```
memory add_port      B    w
memory set_rw_mode B    readafterwrite
memory_port  B    addr  addrb
memory_port  B    di    dib
memory_port  B    we    web    high
memory_port  B    bie   bieb   low

memory add_port      C    r
memory set_rw_mode C    readafterwrite
memory_port  C    do    doc
memory_port  C    addr  addrc
memory_port  C    oe    oec    low

memory generate
```

3.8.6.14 Script Example for a dual-port zrm-based memory model

The following script corresponds to a synchronous, zrm-based memory model with 2 ports (A and B), with the following characteristics:

- The memory size is 1024 words, with 32-bit words.
- Port A is a read-write port, with a 5-cycle output latency the clock for port A is `clka`, active on positive edge; the connected signals are `addr`, `di`, `do`, and `we` is a write enable.
- Port B is a read-write port, with a default output latency; the clock is `clkb`, active on negative edge; the connected signal are `addr`, `di`; `we` is an active-high write enable; `bie` is an active-low bit-enable.

```
memory new memName zrm
memory depth 1024
memory width 32

memory add_port  A    rw
memory set_rw_mode  A    readbeforewrite
memory set_port_latency  A    5
memory_port  A    clk    clka    posedge
memory_port  A    addr    addra
memory_port  A    di      dia
memory_port  A    do      doa
memory_port  A    do_ack  do_acka
memory_port  A    we      wea    low
memory_port  A    be      bea    high

memory add_port  B    rw
memory set_rw_mode  B    readbeforewrite
memory_port  B    clk    clkb    posedge
memory_port  B    addr    addrb
memory_port  B    di      dib
memory_port  B    do      dob
memory_port  B    do_ack  do_ackb
memory_port  B    be      beb    high
memory_port  B    re      reb    low

memory generate
```



3.9 Debug

This section lists what has to be declared in the compilation project to support the appropriate debugging features at runtime.

Some of these debugging features are described in details in separate Application Notes.

3.9.1 Accessing Signals in ZeBu

3.9.1.1 Types of Probes

ZeBu implements different types of probes which have different compilation and runtime constraints:

- **Static Probes:** any internal net of the design can be routed to the top-level interface for high performance debugging, in particular for use with SRAM Trace feature, as described in Section 3.4.7.
All the signals declared as output ports in the DVE file are automatically routed as static probes without any specific declaration. Inserting static probes modifies the netlist therefore it requires a new ZeBu compilation. Note that static probes also impact the FPGA Place and Route process by increasing the number of inter-FPGA signals.
The maximum number of static probes is due to the connection of these probes to the verification environment, because of the size limitation of messages exchanged between the DUT and its environment. According to your use model, the maximum number of static probes varies:
 - When tracing into memory (using SRAM_TRACE driver), you can use up to 4096 single bits, or 128 x 32-bit vectors, or a mix (assuming that a tristate I/O takes twice the size of a binary I/O).
 - With HDL or C/C++ co-simulation, you can use up to ~8100 single-bit signals.
- **Dynamic Probes:** Except in case of simplification by the compiler, all the sequential signals (in particular registers) present in the EDIF netlists are automatically available for selection at runtime without being declared explicitly as probes in the compilation project.
The combinational signals of the design require some specific declaration to be accessed for reading at runtime, as described in Section 3.9.2.
- **Flexible Probes:** Any signal of the design can be declared as a flexible probe in order to provide debugging with similar performance to static probes but without their hardware cost and without recompiling the design to change the active probes. At runtime, flexible probes can be selected on a group-by-group basis.

The declaration of probes is done in one or several Tcl files, which can use the RTL or EDIF paths to instances of the design as described in the following sections. Once

created, these files are added in the **RTL-based compilation scripts** item of the project tree.

Note that adding probes, whatever their type, does not provide the write access, in particular for combinational signals which need additional declaration, with commands described in Sections 3.7.5 (`force`) and 3.7.6 (`force_dyn`).

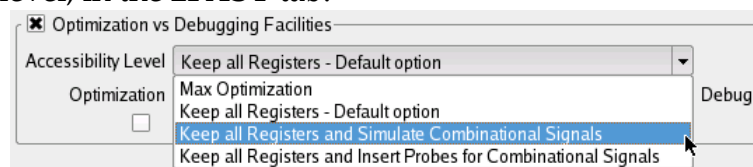
3.9.1.2 Accessing Combinational Signals

Combinational signals of the design can be accessed through different methods:

- Insertion of probes: To have runtime read access to the combinational signals, the ZeBu compiler can add dynamic probes on all these signals in the design, which strongly impacts the size of the design. It is also possible to declare manually in an RTL-based compilation script the signals for which dynamic probes will be added, requiring the mandatory recompilation of the design for any addition/removal of a probe.
- Combinational Signals Accessibility feature (CSA): When synthesizing with **zFAST**, ZeBu offers the capability to simulate the combinational signals of the design so that they can be used easily at runtime for signal monitoring, tracing and waveform dumping. When the CSA feature is activated, the ZeBu runtime database includes additional information so that all the combinational signals can be simulated and displayed at runtime. All the appropriate tools are embedded in the ZeBu software and there is no need for an additional simulator.

Compiling with Access to All Combinational Signals

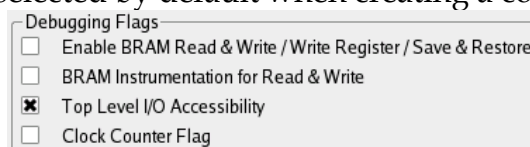
For both probes and simulated signals, the ZeBu compiler settings are available at the RTL group level, in the **zFAST** tab:



In the **Optimization vs Debugging Facilities** frame, select the appropriate value in the **Accessibility Level** selector:

- To access simulated combinational signals, choose **Keep all Registers and Simulate Combinational Signals**.
- To add probes for the combinational signals, choose **Keep all Registers and Insert Probes for Combinational Signals**.

It is important to check that the **Top-Level I/O Accessibility** checkbox is selected in the **Debugging** panel (selected by default when creating a compilation project):





Manual Declaration of Dynamic Probes for Combinational Signals

To provide read access for combinational signals in the same way as for the registers of the design, dynamic probes can be declared in a Tcl file added in the **RTL-based Compilation Scripts** item of the project tree.

Such declaration uses the `-type dynamic` option for `probe_signals`, with the RTL or EDIF path to a combinational signal of the design.

3.9.2 Declaring the Probes in a Tcl File

The probes can be declared in a Tcl file with the `probe_signals` command. This file is used by the synthesis so that all the signals declared as probes are available with their RTL name in the EDIF netlists and in the runtime database.

The following types of probes can be declared with the `probe_signals` command:

- To declare dynamic probes for both registers and combinational signals, the `-type dynamic` option is added. This is the default value when `-type` is not set.
- To declare flexible probes, the `-type flexible` option is added.

The following table lists the options of the `probe_signals` command which can be added for each type of usage:

Table 6: Options for the `probe_signals` Command

Dynamic Probes	Flexible Probes
<code>-type dynamic</code>	<code>-type flexible</code>
<code>-fnmatch</code>	<code>-fnmatch</code>
<code>-wire <edif_signal></code> <code>-wire_file <file></code> <code>-rtlname <rtl_signal></code> <code>-rtlname_file <rtl_file></code>	<code>-wire</code> <code>-wire_file <file></code> <code>-rtlname <rtl_signal></code> <code>-rtlname_file <rtl_file></code>
<code>-instance</code> <code>-instance_file</code> <code>-rtlname <rtl_inst></code> <code>-rtlname_file <rtl_file></code>	<code>-instance</code> <code>-instance_file</code> <code>-rtlname <rtl_inst></code> <code>-rtlname_file <rtl_file></code>
<code>-instance <inst> -wire <signal></code> <code>-inst <rtl_inst> -rtlname <rtl_sig></code>	<code>-instance <inst> -wire <signal></code> <code>-inst <rtl_inst> -rtlname <rtl_sig></code>
<code>-port</code> <code>-port_file</code> <code>-rtlname <rtl_port></code> <code>-rtlname_file <rtl_file></code>	<code>-port</code> <code>-port_file</code> <code>-rtlname <rtl_port></code> <code>-rtlname_file <rtl_file></code>
<code>-module</code> <code>-module_file</code>	<code>-module</code> <code>-module_file</code>
<code>-filter_out</code> <code>-select</code> <code>-exclude</code>	<code>-filter_out</code> <code>-select</code> <code>-exclude</code>
<code>-size_lwr_than</code> <code>-size_gtr_than</code> <code>-depth</code>	<code>-size_lwr_than</code> <code>-size_gtr_than</code> <code>-depth</code>
<code>-hier_sep</code>	<code>-hier_sep</code>
	<code>-group</code> <code>-clock_name</code> <code>-fifo</code>
	<code>-enable_wire</code> <code>-enable_port</code> <code>-enable_rtl <rtl_signal></code> <code>-enable_rtl <rtl_port></code>



The `probe_signals` command is also useful to declare signals which are used for other debugging features:

- To connect manually signals for DPI calls, signals can be declared with `-type ccall`. Note that this does not apply to zDPI feature.
- To connect signals of the design for the Value Change feature with `-type vc`.

C-Calls	Value Change
<code>-type c-calls</code>	<code>-type vc</code>
<code>-fnmatch</code>	<code>-fnmatch</code>
<code>-wire</code> <code>-wire_file</code> <code>-rtlname <rtl_signal></code> <code>-rtlname_file <rtl_file></code>	<code>-wire</code> <code>-wire_file</code> <code>-rtlname <rtl_signal></code> <code>-rtlname_file <rtl_file></code>
<code>-instance</code> <code>-instance_file</code> <code>-rtlname <rtl_inst></code> <code>-rtlname_file <rtl_file></code>	<code>-instance</code> <code>-instance_file</code> <code>-rtlname <rtl_inst></code> <code>-rtlname_file <rtl_file></code>
<code>-instance <i> -wire <s></code> <code>-inst <rtl_i> -rtlname <rtl_s></code>	<code>-instance <i> -wire <s></code> <code>-inst <rtl_i> -rtlname <rtl_s></code>
<code>-port</code> <code>-port_file</code> <code>-rtlname <rtl_port></code> <code>-rtlname_file <rtl_file></code>	<code>-port</code> <code>-port_file</code> <code>-rtlname <rtl_port></code> <code>-rtlname_file <rtl_file></code>
<code>-module</code> <code>-module_file</code>	<code>-module</code> <code>-module_file</code>
<code>-filter_out</code> <code>-select</code> <code>-exclude</code>	<code>-filter_out</code> <code>-select</code> <code>-exclude</code>
<code>-size_lwr_than</code> <code>-size_gtr_than</code> <code>-depth</code>	<code>-size_lwr_than</code> <code>-size_gtr_than</code> <code>-depth</code>
<code>-hier_sep</code>	<code>-hier_sep</code>
<code>-group</code> <code>-clock_name</code>	<code>-group</code> <code>-clock_name</code>
<code>-rank</code>	
<code>-scope</code>	
	<code>-edge</code> (of the signal which is selected for detection)

For any type of probe declared with the `probe_signals` command, the declaration of the signals can be done either signal-by-signal or on a pattern matching basis with the `-fnmatch` option.

When `-fnmatch` option is added for the command, any wire, instance, module or port in the command can be a pattern, using `*` (to replace one or more characters) or `?` (to replace one character).

The declaration of signals can be done either explicitly (`-wire`, `-module`, `-instance`, `-port` options) or in a separate file which lists the wires, instances, modules or ports (`-wire_file`, `-module_file`, `-instance_file`, `-port_file` options). When `-fnmatch` option is used for declaration with an additional file, the pattern-matching is done on the elements listed in the file.

As described in Section 3.7.2, the probes can be declared with RTL or with EDIF paths (different options are expected according to the type of declaration).

Any line starting with a `#` in the file is ignored.



3.9.3 SystemVerilog Assertions (SVAs) with zFAST

When synthesizing with **zFAST**, ZeBu supports SystemVerilog Assertions (SVA) for compilation and runtime. This feature is fully described in a dedicated Application Note (AN028 - *SystemVerilog Assertion (SVA) for ZeBu*).

SystemVerilog Assertions are compliant with the IEEE SystemVerilog Standard (IEEE Std. 1800™-2005). The supported subset of the language supported by ZeBu is described in AN028.

For SVA support in ZeBu, compilation options can be set in **zCui**:

- Selection of the synthesized assertions in the **SVA** tab for the RTL group.
- Activation of a trigger mechanism:
 - By default, a trigger is inserted to stop emulation in the cycle for assertions with a `$fatal` action block.
 - This trigger insertion can be disabled by user when FPGA filling-rate is critical.
- Selection of the runtime information level:
 - Reports start and failure time of assertions (default)
 - Reports only failure time of assertions for runtime performance optimization.

The support of SystemVerilog Assertions with ZeBu requires a specific license feature which can be purchased from your EVE representative.

3.9.4 Summary of the Compilation Settings for Debug

When bringing up the design, it is recommended to check that the following options are correctly set in the **zCui** compilation project:

- SystemVerilog Assertions, probe declaration with RTL names and simulated combinational signals are supported only when there is only one RTL group for the project.
- Declare the probes in one or several files (**RTL-based Compilation Scripts** item in the project tree).
- In the **zFAST** or **Synthesizer** tab of the RTL group, choose the most appropriate value for the **Accessibility Level**.
- Check that the appropriate signals are declared in the DVE file for static probes purposes.
- Check that the **Memory I/O Accessibility on all zMem models** checkbox (**Memory Sources** panel) is selected.

Check that the **Debugging Flags** are correctly set (in the **Debugging** panel).



3.9.5 Detection of Combinational Loops

Because combinational loops in the design make the design unpredictable at runtime, the ZeBu compiler provides the capability to detect them. This detection can be activated upon user request by adding the following command in an additional command file declared through the **Advanced** → **Build System Parameters** frame:

- To have the combinational loops reported after compilation:

```
loop report
```

- To add the appropriate resources for runtime debugging:

```
loop runtime_detect
```

When some combinational loops are detected they are reported in dedicated sections of the compilation reports:

- Inter-zCores loops are reported by the system-level compiler (section “14. Inter-partition combinational loops checks”). A warning is displayed in the log file of the **Build System** step:

```
### warning in zTopBuild [ZTB0298W] : 2 combinational loops have been found in netlist.
```

- Combinational loops which are inside a zCore are reported by the corresponding zCore-level compiler (section “8.5. Combinational loop reporting” and section “8.6. Combinational loop instrumenting”). A warning is displayed in the log file of the corresponding **Build zCore** step:

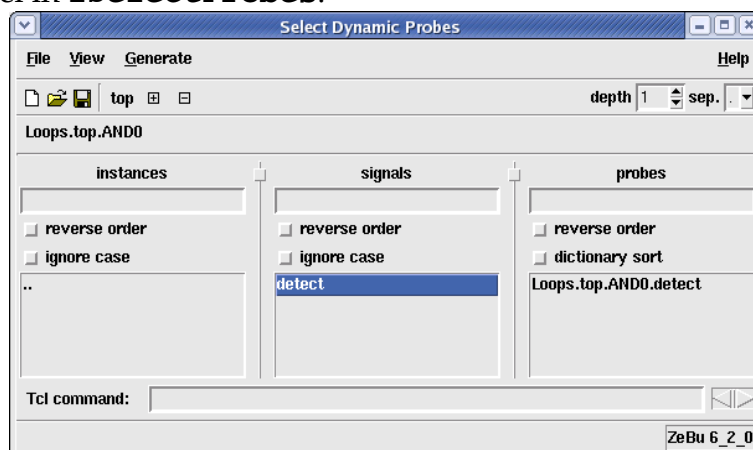
```
### warning in zCoreBuild [KBD1628W] : 3 combinational loops have been found in netlist.
```

Note that for designs with many combinational loops, it may be of interest to declare a threshold so that the Zebu compiler stops its analysis for combinational loops. For that purpose, the `-max_loop_count <n>` option can be added for the `loop` command:

```
loop report -max_loop_count 10000
```

In such a case, the compiler reports only include information for the `<n>` first loop and only these `<n>` loops are available for runtime debugging.

When the runtime debugging is activated, a dedicated register is available as a dynamic probe for each combinational loop in the `Loops.<path_to_loop>` hierarchical level in **zSelectProbes**:





In the Loops hierarchical level, each loop is identified with the same name as in the compilation reports (`top.AND0` here) and the `detect` register is available: it switches to 1 when the corresponding combinational loop oscillates at runtime. In the testbench, it is possible to iterate through the detection registers in order to check that a combinational loop is not the source of design instability or runtime misbehavior.

Once the detection registers have toggled to 1, they are not automatically reset to 0. This can be done manually for all the detection registers at once, by modifying from the testbench the value of the `zdve_zloops_reset_dve_reg` register which is created automatically and is similar to a DVE register:

```
Signal* rst = z->getSignal("zdve_zloops_reset_dve_reg");
```

With this example, `rst` has to be set to 1 to reset the detectors and then set to 0 to go on with detection of oscillations.

The C++ and C runtime API also provides the corresponding methods and functions to control the detection of oscillating combinational loops. These are described in `LoopDetector.hh` (C++ API) and `ZEBU_LoopDetector.h` (C API). These files are automatically included with `libZebu.hh/libZeBu.h`.

4 Compiling with zCui

4.1 Compilation Settings

4.1.1 Declaring the Hardware Configuration File

The ZeBu compiler needs information about the hardware configuration of your ZeBu system to proceed with automatic clustering, zrm memory allocation and to provide a runtime database that matches the actual state of your system (in particular for FPGA faulty pins).

Such information is transmitted to the ZeBu compiler in a Tcl configuration file which is created during installation of your ZeBu system (usual name is `zse_configuration.tcl`).

Sample configuration files are available in the ZeBu software package and can be used to initiate a compilation project. Such files can be found in the `$ZEBU_ROOT/etc/configurations` directory.

However, it is strongly recommended to create a configuration file which actually matches the targeted ZeBu-Server system with **zConfig**, as described in the *[ZeBu-Server Installation Manual](#)*. This specific file is mandatory to proceed with emulation runtime.

This configuration file is declared in the **Back-End → Configuration** panel:

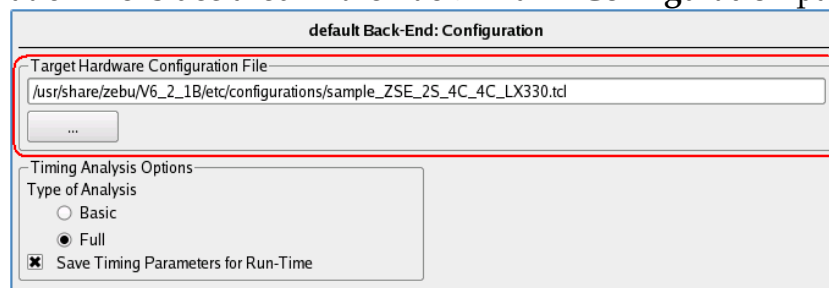


Figure 50: Declaring the Hardware Configuration File (example with sample file)

4.1.2 Options for storage of FPGA P&R intermediate files

By default, all the intermediate files generated during FPGA Place and Route, in particular the EDIF netlists, are compressed during the process. This default behavior balances the disk space usage and the incrementality of the compilation since the required intermediate files are available when relaunching the Place and Route for a given FPGA.

Different options can be selected in the **FPGA P&R File Policy** frame (**Back-End → FPGA P&R** panel), in order to keep all the intermediate files without compression (choose **Regular**) or to delete all the files (choose **Suppress**):

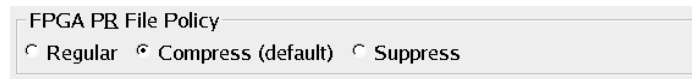


Figure 51: FPGA P&R File Policy

Note that deleting the intermediate files may impact the future compilations if the FPGA P&R settings are modified (all the intermediate steps have to be re-processed).

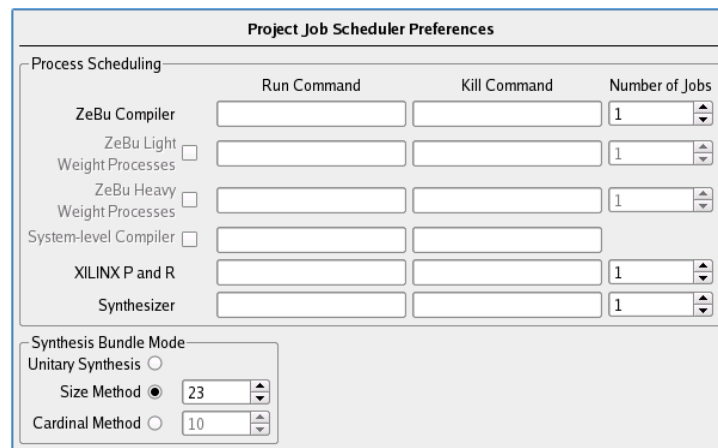
4.1.3 Configuring the Compiler for Job Scheduling

When compiling for ZeBu, the compilation tasks have different constraints in terms of PC resources:

- Synthesis tasks are constrained by the availability of licenses.
- FPGA Place & Route tasks require a high amount of memory resources.
- Among the other compilation tasks, there are also major differences:
 - Script creation and result analysis tasks are very light and can be launched on the PC where **zCui** is running.
 - The back-end compilation tasks which process the netlists (in particular the system-level compiler) and FPGA Place and Route tasks need a high amount of memory.

By default, the ZeBu compiler processes the compilation on the PC from which it was launched. However, it is possible to use an external task scheduler such as Sun Grid Engine or Platform LSF in order to share the load among a PC farm.

In the **Preferences** → **Job Scheduling** panel, you can configure different scheduling options for these tasks:



Project Job Scheduler Preferences			
Process Scheduling			
	Run Command	Kill Command	Number of Jobs
ZeBu Compiler			1
ZeBu Light Weight Processes			1
ZeBu Heavy Weight Processes			1
System-level Compiler			
XILINX P and R			1
Synthesizer			1

Synthesis Bundle Mode

Unitary Synthesis ☐

Size Method ☒ 23

Cardinal Method ☐ 10

Figure 52: Preferences → Job Scheduling Panel

Specific scheduling parameters are expected for FPGA Place and Route and for synthesis. By default, all the other compilation tasks use the same scheduling parameters declared in the **ZeBu Compiler** line. It is possible to declare various options:

- To run the script creation and result analysis tasks with specific options, specific options can be set in the **ZeBu Light Weight Processes** line. Typically, to run such tasks locally, the **Run Command** and **Kill Command** fields are left blank.



- To use specific options for the back-end compilation tasks which manipulate the netlist, the checkbox for **ZeBu Heavy Weight Processes** should be selected and the appropriate commands added in the **Run Command** and **Kill Command** fields. If the checkbox is not selected, the **ZeBu Compiler** commands are used.
- To use specific options for the system-level compiler, the checkbox for **System-level Compiler** should be selected and the appropriate commands added in the **Run Command** and **Kill Command** fields. If the checkbox is not selected, the **ZeBu Heavy Weight Processes** commands are used (if set) or the **ZeBu Compiler** commands.

For each type of task, you can declare the remote commands to launch and to kill the corresponding jobs, as well as the maximum number of jobs that will be processed in parallel:

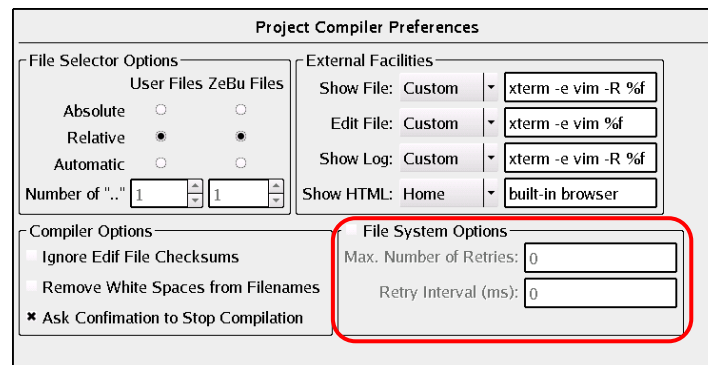
- To launch and control a task on the PC where **zCui** is running, the corresponding field should be left blank. In such a case, the **Max Number of Jobs** selector should be set to the number of available processors of the PC for efficiency purposes (launching more synthesis and FPGA P&R jobs in parallel would drastically impact the compilation).
- To launch and control a task on a PC farm, the content of the **Run Command** and the **Kill Command** fields depends on the load sharing tool. The following recommendations apply for Sun Grid Engine and Platform LSF, which are the most popular distribution software in use with ZeBu:
 - For Sun Grid Engine:
In the **Run Command** field:
`| qrsh`
 - In the **Kill Command** field:
`| qdel`
 - For Platform LSF with blocking jobs and a non-interactive queue:
In the **Run Command** field:
`| bsub -K`
 - In the **Kill Command** field:
`| bkill`
 - For Platform LSF with blocking jobs and an interactive queue:
In the **Run Command** field:
`| bsub -Is`
 - In the **Kill Command** field:
`| bkill`
 - Additional options may be necessary for these commands to match the constraints of the IT configuration, in particular to target only compilation PCs with 64-bit operating systems and to manage priorities.

- The **Max Number of Jobs** selector should be set to an appropriate value which takes into account the number of processors in the PC farm (which is the maximum number of jobs for efficiency purposes) and the acceptable load of the PC which runs the load sharing tool.

4.1.4 Retry mechanism for NFS failures

When the ZeBu compiler uses a farm of PCs, the NFS protocol is used when writing files. Since **zCui** checks the completion of each task through the availability of the corresponding output files, a slow NFS system may cause some unexpected compilation errors.

These “Missing Inputs” errors can be avoided by modifying the default settings of **zCui** in order to check several times if a file is available and/or increase the delay before checking for availability. The number of attempts to reach a file before returning an error and the delay (in ms) between two attempts can be set in the **Preferences → Compiler → File System Options** frame:




The screenshot shows the 'Project Compiler Preferences' dialog box. It has several sections: 'File Selector Options', 'External Facilities', 'Compiler Options', and 'File System Options'. The 'File System Options' section is highlighted with a red rectangle. It contains two input fields: 'Max. Number of Retries' and 'Retry Interval (ms)', both set to 0.

The values for these fields must be chosen with much care since they impact the delay of error messages for files which are actually not available (delay = max nb retries * interval).

The **zCui** full log file includes information which may help to determine appropriate values:

```
# Network File System Stats: Number of Retry = 0 ; Number Of Failure = 0 ; Average Number of Retries = 0 ; Maximum Number Of Retries = 0
```

- The Maximum number of Retries value is a correct basis for delay, but a security margin is recommended (5ms seems a reasonable margin).
- Average number of Retries: if this number is high (it should stay below 500), it means that interval should be increased in order to reduce the number of system calls).

The **zCui** full log is available in the **View** menu or by clicking the following icon: .

Note:

zCui sometimes fails to detect that a file has changed because the file stamp is not updated between two consecutive compilation tasks. In such a case, you should check the **Tell Why?** contextual menu of the failing compilation task: if it shows “same date”, it may be caused by an incrementality issue.

4.1.5 Compiling a design for relocation

By default, the design is compiled to use all the resources of the ZeBu system (all the FPGAs of all the FPGA modules in the system can be used for the design and for routing purposes). In a multi-user environment, it is recommended to compile for a minimal subset of FPGA modules in order to support relocation at runtime.

Before compiling for relocation, the hardware configuration file must be generated with **zConfig** using `-merge` option, as described in the [ZeBu-Server Installation Manual](#).

Furthermore, the size of the design has to be estimated in order to declare the necessary FPGA modules so that the compiler has enough resources to implement the design. For each necessary FPGA module, the following command should be added in the additional command file declared in the **Advanced** → **Build System Parameters** frame:

```
use_module -loc <module_id>
```

Where `<module_id>` is a string, typically `Ux.My` (or `UNITx.MODy`) which is the identifier of the targeted FPGA module in the ZeBu-Server configuration.

Once the design is compiled, the number of declared FPGA modules can be potentially optimized by considering the design statistics.

4.2 Compiling the Project

4.2.1 Choosing the compilation target

You can choose to process different parts of the compilation project by changing the value in the **Target** selector of the **Compiler** toolbar (the default value when launching **zCui** is **Default Back-End**):

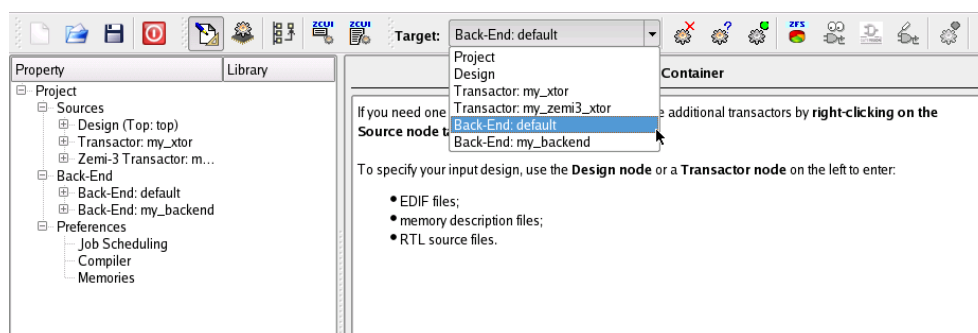


Figure 53: Choosing the compilation target

The number of lines in this box changes with the architecture of the compilation project: the **Project**, **Design** and **Default Back-End** items are always present; one item is added for each additional transactor declared in the **Project** → **Sources** and for each additional back-end.

The selected target impacts the behavior of the **Clean**, **Evaluate** and **Make** operations:

- **Project** item: removes all the results for all front-end and back-end previous compilations; available only for the **Clean** operation.
- **Design** item: launches or clears only the results of the synthesis of the source files in the **Design** item. Using this target is mandatory to browse the EDIF netlist of all the design with **zBrowser** or with GateVision.
- **Default Back-end** item: evaluates or launches the compilation of the default back-end and the synthesis if some dependencies have changed; clears only the results of the default back-end
- **Transactor: xxx** item: same as **Design** item, but for the source files of the transactor.
- **Back-End: xxx** item: same as **Default back-end**, but for the corresponding back-end.

4.2.2 Launching the Compilation

The compilation process can be launched from the **Compiler** toolbar or from the **Tools** Menu:

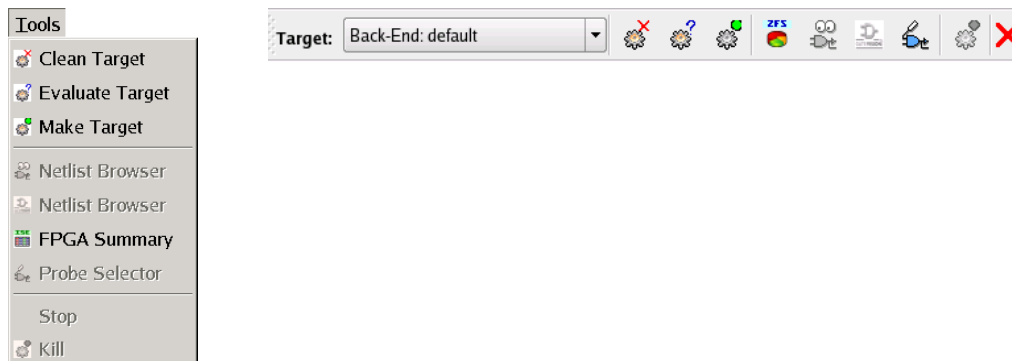


Figure 54: Tools Menu and Compiler Toolbar

The following operations can be done:

- **Clean** (🗑️): removes the resulting files of the previous compilation but the log files are kept.
- **Evaluate** (❓): checks the status of the compilation target (requires the same write access as to actually compile the design).
- **Make** (🔧): starts the compilation of the target (launches only the tasks for which some dependencies have been modified).

4.3 Monitoring the Compilation

4.3.1 During Compilation

When the compilation is launched, **zCui** automatically switches to the **Compilation** view and displays the status of the compilation steps in the **Task Tree** pane:

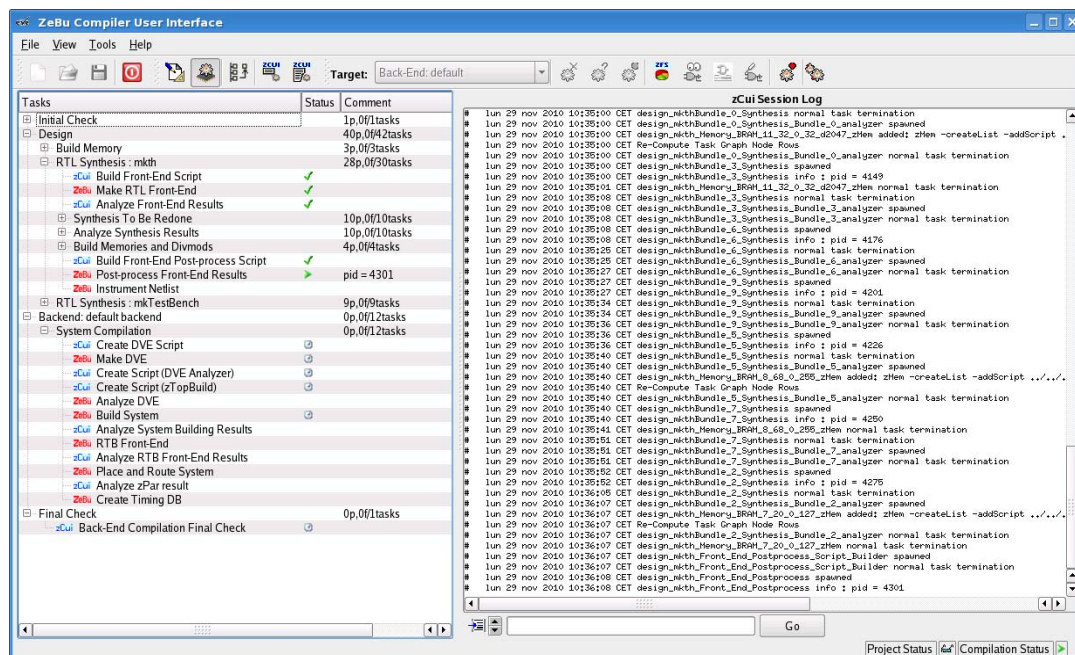









Figure 55: Task Tree Pane during Compilation

For each compilation task, the **Task Tree** pane shows the current status:

-  Pending task (not yet launched by **zCui**).
-  Task in progress.
-  Task successfully finished.
-  Completed task; failed.
-  Task with missing inputs (only for the **Evaluate** operation).
-  Group in progress but 1 task (or more) of the group failed.
-  Cancelled pending task (in case of PARFF, as described in Section 5.4.2).

Note that the same icons are used for the **Compilation Status** icon (bottom-right corner).

In the **Task Tree** pane, the compilation tasks are gathered and for each group of tasks, a global status is displayed in the **Comments** column with the number of passed and failed tasks. Once a group has successfully completed, it is reduced to only one line to have a better display in the **Task Tree** pane.

4.3.2 Successful Compilation

When the compilation is successful, it may be of interest to have a look at the following points, in particular when targeting optimization of runtime performance.

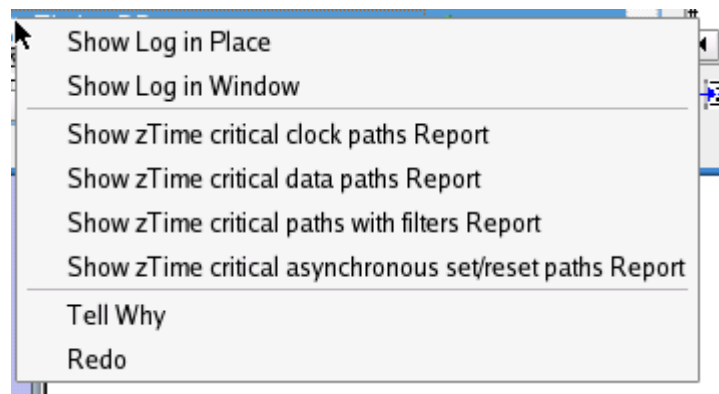
- Timing analysis results:

The log file of the **Create Timing DB** step reports all the appropriate information about the estimated driverClock frequency and clock skew time.

```
#-----#
# step REPORT : Longest inter-fpga filter path delay is : 1166 ns
# step REPORT : Longest inter-fpga clock path delay is : 1166 ns
# step REPORT : zClockSkewTime run-time parameter is : 1186 ns
# step REPORT : Max pessimistic delay (routing and clock skew) is : 2602 ns
# step REPORT : Critical routing path delay : 1416 ns
# step REPORT : . Constant part : 250 ns
# step REPORT : . Multiplexed part : 1166 ns
# step REPORT : Longest memory period is : 341 ns
# step REPORT : No flexible probes found
# step REPORT : Driver clock frequency is limited by routing paths
# step REPORT : The theoretical frequency using default settings and
#                  ignoring clock skew is 706 KHz
#-----#
```

By default, the estimated parameters (driverClock frequency, zClockSkewTime and zClockFilterTime) are automatically taken into account by the ZeBu runtime environment.

When targeting performance optimization, several reports of the static timing analysis task should be analyzed. These HTML reports can be accessed from the contextual menu of the **Creating Timing DB** step in the **Compilation** view:



For details, you should refer to the dedicated Application Note (AN017).

4.4 Other Options

4.4.1 Changing the editor for source files and logs

The tools to view/edit the source files and logs can be changed in the **Project Compiler Preferences** panel:

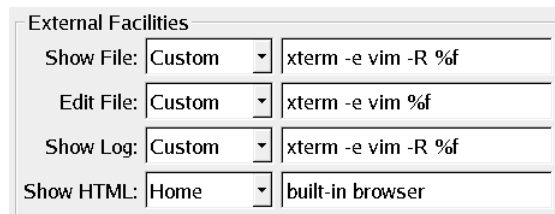


Figure 56: External Facilities frame in Preferences → Compiler panel

For each type of file, different choices are proposed:

- vi and emacs editors are proposed for source files and text format log files.
- **zCui** built-in browser (**Home** value), Konqueror and Firefox are proposed for HTML log files.
- User-defined tools can also be declared (**Custom** value) with their specific command line.

Note that %f is mandatory in the command line to have the appropriate file opened.

4.5 zCui Archiver Feature

The **zCui** Archiver feature is available in 2 different modes:

- Batch mode: for runtime files only.
- Graphical mode: for runtime files as well as project sources and debug files.

4.5.1 Archival in Batch Mode

In batch mode, only back-end files destined for emulation runtime can be archived. To archive other files such as project sources and debug files, see Section 4.5.2.

4.5.1.1 To archive the default back-end only:

To store the runtime files for the default back-end project, you should use one of the following commands.

```
$ zCui -p <project>.zpf -n -t <def_backend> (with no file extension)
$ zCui -p <project>.zpf -n -t <def_backend>.tgz
$ zCui -p <project>.zpf -n -t <def_backend>.tar.gz
$ zCui -p <project>.zpf -n -t <def_backend>.tbz2
$ zCui -p <project>.zpf -n -t <def_backend>.tar.bz2
```

Where:

- -n is a mandatory parameter to use the **zCui** batch mode (stands for “no GUI”).
- <def_backend>: any name to identify the default back-end archive.
- tbz2 and tar.bz2 allow for more compression than tgz and tar.gz



As a result, 2 tar files are generated:

- `<def_backend>.tgz` (or `tar.gz`, `tbz2`, `tar.bz2`): Archived files (with dereferenced links)
- `<def_backend>_links.tgz` (with `tgz` extension only): No dereferenced links (without archived files)

If you specify...	...then the following files will be generated:
<code><def_backend></code> (with no file extension)	<code><def_backend>.tgz</code> <code><def_backend>_links.tgz</code>
<code><def_backend>.tgz</code>	<code><def_backend>.tgz</code> <code><def_backend>_links.tgz</code>
<code><def_backend>.tar.gz</code>	<code><def_backend>.tar.gz</code> <code><def_backend>_links.tgz</code>
<code><def_backend>.tbz2</code>	<code><def_backend>.tbz2</code> <code><def_backend>_links.tgz</code>
<code><def_backend>.tar.bz2</code>	<code><def_backend>.tar.bz2</code> <code><def_backend>_links.tgz</code>

4.5.1.2 To archive the default back-end along with other back-ends:

```
$ zCui -p <project>.zpf -n -t <def_backend>.tgz  
-b <my_backend_X> -t <my_backend_X>.tgz
```

Where `<my_backend_X>` is any back-end other than the default back-end

In the example above, `tar.gz`, `tbz2`, and `tar.bz2` extensions can also be used and the same remarks as above apply.

Note: If `-t <my_backend_X>` is missing in the command, then only the default back-end is archived. In other words, `<my_backend_X>` is but not archived, and the following files are not generated: `<my_backend_X>.tgz` and `<my_backend_X>_links.tgz`

The following examples show that user can issue a single command to compile and/or archive several back-ends. In all examples, the default back-end is always compiled AND archived as `default_backend.tgz`.

Example 1:

```
$ zCui -p project.zpf -n -t def_backend.tgz \  
-b my_backend_1 -t archive_1.tgz \  
-b my_backend_2 -t archive_2.tgz \  
-b my_backend_3 -t archive_3.tgz
```

`backend_1`, `backend_2`, and `backend_3` are archived respectively as `archive_1.tgz`, `archive_2.tgz`, and `archive_3.tgz`.

Example 2:

```
$ zCui -p project.zpf -n -t def_backend.tgz \  
-b my_backend_1 -b my_backend_2 -t archive_1.tgz
```

`backend_1` is only compiled (it is not archived); `backend_2` is archived as `archive_1.tgz`.

Example 3:

```
$ zCui -p project.zpf -n -t def_backend.tgz \
-b my_backend_2 -t archive_1.tgz -t archive_2.tgz
```

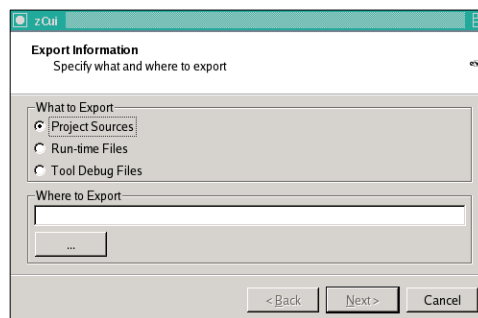
backend_2 is archived as archive_1.tgz and archive_2.tgz (archive_1.tgz and archive_2.tgz are identical).

4.5.2 Archival in Graphical Mode

In graphical mode, user can archive back-end files destined for emulation runtime as well as project sources and debug files (runtime, debug, log). The list of archived files depends on the user-specified compilation options and is displayed in the last **Export Information** window.

Whatever the type of files you want to archive, the following first steps are identical:

1. In the **zCui** menu bar, select **File** → **Archive**.
2. In the **Export Information** window:
 - a. Select **Project Sources**, **Runtime Files**, or **Tool Debug Files**.
 - b. Type the name (and path) of the archive file in the **Where to Export** field or click the [...] button to select an existing archive file.



The table below shows what file extensions are generated with respect to the file extension declared in the command:

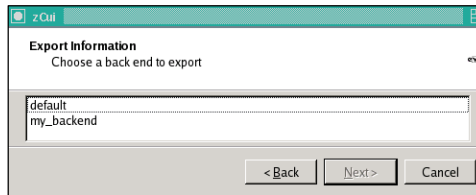
If you declare...	...then the following files will be generated:
<my_backend> (with no file extension)	<my_archive>.tgz <my_archive>_links.tgz
<my_archive>.tgz	<my_archive>.tgz <my_archive>_links.tgz
<my_archive>.tar.gz	<my_archive>.tar.gz <my_archive>_links.tgz
<my_archive>.tbz2	<my_archive>.tbz2 <my_archive>_links.tgz
<my_archive>.tar.bz2	<my_archive>.tar.bz2 <my_archive>_links.tgz

4.5.2.1 Archival of Project Sources

1. Select **Project Sources** in the **Export Information** window, specify the archive file name and click **Next**.
2. In the next **Export Information** window, click the **Archive** button.
3. Wait until the progress bar has reached **100%** and click the **Finish** button.

4.5.2.2 Archival of Runtime Files

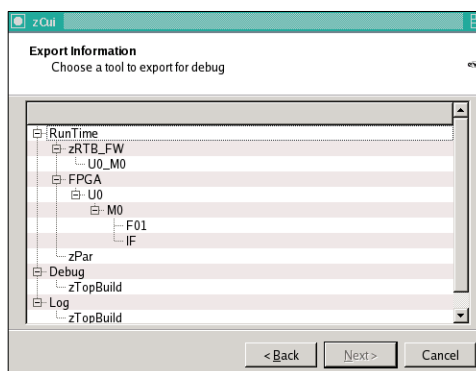
1. Select **Runtime Files** in the **Export Information** window, specify the archive file name and click the **Next** button.
2. In the next **Export Information** window, select the back-end you want to archive (**default** for default back-end, or any other back-end in the list) and click the **Next** button. Note that only one back-end can be selected in an archive.



3. In the next **Export Information** window, click the **Archive** button.
4. Wait until the progress bar has reached **100%** and click **Finish**.

4.5.2.3 Archival of Tool Debug Files

1. Select **Tool Debug Files** in the **Export Information** window, specify the archive file name and click the **Next** button.
2. In the next **Export Information** window, select the back-end you want to archive (**default** for default back-end, or any other back-end in the list) and click the **Next** button. Note that only one back-end can be selected in an archive.
3. In the last **Export Information** window, zoom in on the type of tool debug files you want to archive and click the **Next** button. Please note that multiple selections are not allowed.



4. In the next **Export Information** window, click the **Archive** button.
5. Wait until the progress bar has reached **100%** and click **Finish**.

5 Advanced Compilation

5.1 Retargeting from an ASIC synthesizer

The ASIC synthesis flow does not take advantage of specific Xilinx resources such as multipliers or multiplexers. This usually results in a less efficient utilization of the Xilinx resources, such as LUTs and multiplexers, and in an apparent lower capacity of the Xilinx devices.

However, the EDIF netlist of a design synthesized by an ASIC synthesizer can be ported to ZeBu via an intermediate generic library such as the GTECH library ("gtech") or by developing a dedicated translation library to generate EDIF source files for the ZeBu compilation flow.

5.1.1 Retargeting a Design to GTECH

The simplest way of converting an ASIC design for ZeBu consists of retargeting the ASIC design onto an intermediate generic library such as the GTECH library, supported by most ASIC synthesizers.

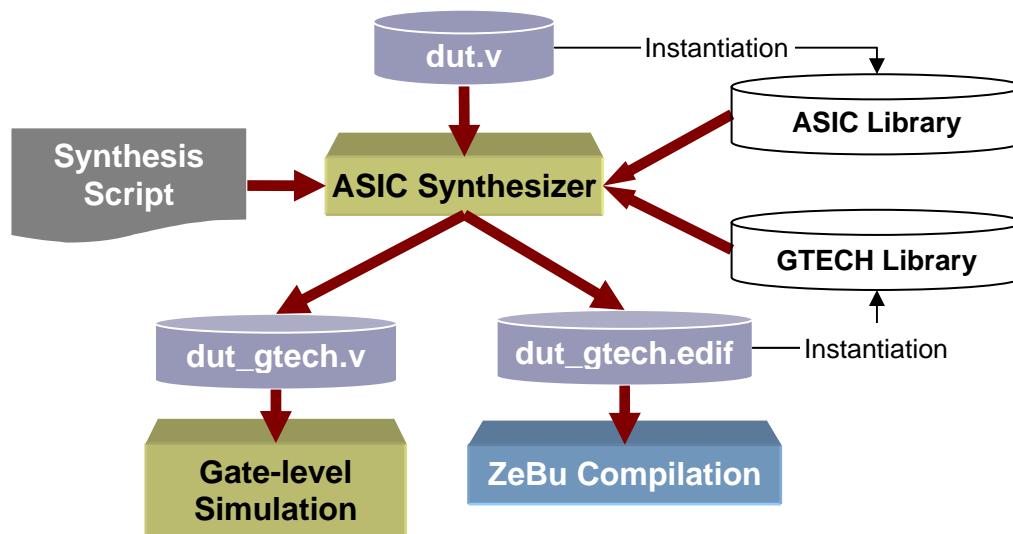


Figure 57: Retargeting an ASIC Design for ZeBu

The following script shows the Design Compiler commands to retarget the whole ASIC design with the GTECH library, generating both EDIF and Verilog netlists:

```

dc_shell-t > read_verilog <dut.v>
dc_shell-t > set target_library "gtech.db"
dc_shell-t > set link_library "ASIC_Library.db"
dc_shell-t > current_design <top_dut>
dc_shell-t > compile -area_effort none -map_effort low -no_design_rule
dc_shell-t > set edifout_netlist_only true
dc_shell-t > write -format edif -output <dut_gtech.edif>
dc_shell-t > write -format verilog -output <dut_gtech.v>
  
```

The resulting EDIF netlist (<dut_gtech.edif> in the above example) is declared as an EDIF source file for the ZeBu compiler. The conversion library

(\$ZEBU_ROOT/etc/libGtechZebu.edif.gz) must also be declared as an EDIF source file (compressed EDIF file):



Figure 58: Declaring the source files when retargeting with GTECH

Note that the Verilog output cannot be used for synthesis in the ZeBu compilation flow.

5.1.2 Flattening the Netlist

When a translation library has been merged in the design, it can optionally be flattened to accelerate the ZeBu compilation and make the runtime database easier to use (the additional hierarchical level added for the library is removed).

For that purpose, the following command is added for the system-level compiler in the **Netlist Edition File (zCore Definition panel)**:

```
flatten -cell <cell_name> [-fnmatch]
```

Where <cell_name> is a string which is either the exact name of a cell or a pattern-based string when -fnmatch option is added (in such a case, <cell_name> includes * for several characters and/or ? for a single character).

Example:

To flatten the entire GTECH library (all gtech_* cells), the following command should be added:

```
flatten -cell "gtech_*" -fnmatch
```

5.1.3 Creating a Dedicated Translation Library

The dedicated translation library offers a direct method to map an ASIC design to the Xilinx technology. For this purpose, each of the ASIC primitives must be described in terms of Xilinx primitives.

The creation of a dedicated translation library is labor intensive, but it only needs to be done once for each standard cell library related to the ASIC target technology. This can be done manually or can be partially automated.

Once made available, the translation library and the design netlist files can be merged with the ZeBu compiler to obtain the design mapped for the Xilinx Virtex technology.



5.1.3.1 Manually Creating a Translation Library

Typically, a translation library takes from one week to one month to develop manually, depending on the number of primitives to be translated. Development of a translation library has to be done once for each target standard cell library.

The process consists of creating a VHDL or Verilog file that describes, for each primitive of the ASIC library, an implementation in terms of Xilinx primitives. The list of primitives available for the Virtex technology can be found in the Xilinx documentation.

The VHDL/Verilog file can be synthesized with your ASIC synthesizer where the Xilinx primitives are considered as blackboxes. There is no target library during this synthesis step and the goal is to generate an equivalent EDIF netlist, which is the translation library.

5.1.3.2 Automatically Creating a Translation Library Using GTECH

One way to automatically create the translation library is to map the target ASIC library onto the GTECH technology, and then use the GTECH translation library to remap the design onto the Xilinx technology. The following is an overview of the method:

1. For each ASIC primitive, create a primitive wrapper (design) that instantiates the primitive.
2. Synthesize all wrappers created during step 1 using the ASIC synthesizer, the `gtech.db` file as the `target_library`, and the ASIC library as the `link_library`.
3. Merge all the resulting files generated during step 2 with the `gtech.edif` file to create the translation library. Note that it may be necessary to rename the primitive wrappers to match the original ASIC primitive names.

The result is a hierarchical translation library that contains the ASIC primitives ultimately described in terms of Xilinx primitives.



5.2 Advanced Clock Modeling

5.2.1 ZeBu-Server Uncontrolled Clocks

In addition to the controlled clocks driven by the ZeBu clock generator, ZeBu-Server can use input clocks coming for the Smart Z-ICE or Direct ICE interfaces. It also supports frequency synthesizers which are recommended for free-running clocks in prototyping mode.

Glitch filtering and synchronous delay insertion is not applicable given the asynchronous nature of uncontrolled clocks.

5.2.1.1 Primary Clocks coming from Direct ICE or Smart Z-ICE

The primary clock inputs coming from a Direct ICE target or from the Smart Z-ICE target are declared in the DVE file as instantiations of the `zIceClockPort` macro:

```
zIceClockPort <clock_ID> (.clock (<clock_name>))
```

Such clocks cannot be declared in **zCui** graphical interface and cannot be automatically added when generating a DVE file template.

Further details on such clocks are available in the [ZeBu-Server Direct ICE Manual](#) and [ZeBu-Server Smart Z-ICE Manual](#).

5.2.1.2 Frequency Synthesizers for Prototyping

For each FPGA module of the ZeBu-Server unit, two frequency synthesizers are available, programmable between 3.125 and 450 MHz.

Such synthesizers can be used to generate asynchronous clocks to drive DCM in the design FPGAs or to clock the trace memory asynchronously. These synthesizers can clock only instances in the design FPGAs of the same module; in most cases, such clocks are used in one FPGA only because of the frequency constraints.

Such a frequency synthesizer is declared in the DVE file as an instantiation of the `zClockPort` macros, with a maximum frequency constraint for compilation:

```
zClockPort <my_ClockPort_inst> (  
    .clock( <my_clock> )  
);  
defparam <my_ClockPort_inst>.frequency = <my_max_freq>;  
defparam <my_ClockPort_inst>.synthesis = true;
```

Where `<my_max_freq>` is the maximum frequency constraint, declared as a double-quoted string with value (integer) and unit (kHz or MHz, case insensitive). Example: `<my_freq>="54000Khz"`.

Their actual frequency can be modified at runtime in the `designFeatures` file, as long as it is lower than `<my_max_freq>`.

The skew control for such clocks can be done by selecting **Asynchronous** in the **Skew Offset** frame of the **Clock Handling** panel. The value of the inserted delay is set in ns in the selector; it should be set between 100 and 640.



5.2.2 Configuring the allocation of clock resources for user clocks

5.2.2.1 Number of Clock Buffers

By default, the ZeBu compiler allocates 6 clock buffers (BUFGs) for user clocks so that they are mapped automatically to the low-skew routes in the FPGAs. The clocks with the highest fanout values are mapped to these resources in priority.

The available number of such buffers for user clocks can be increased (maximum is 12) if more user clocks have to be mapped with BUFGs or decreased in case of FPGA compilation failures.

For that purpose, the following command should be added in the additional command file in the **Advanced** → **Build System Parameters** frame:

```
set_config -bufg <n>
```

Where <n> is an integer (minimum is 3; maximum is 12).

5.2.2.2 Settings for Clock Buffers Allocation

In order to optimize the allocation of clock buffers, the fanout criteria becomes stronger when the number of BUFGs increases (the minimum fanout to map a clock with a clock buffer increases).

By default, the minimum fanout for the first BUFG allocation is 0 (which means that the highest fanout clock in the design will use a BUFG in any case). After each BUFG allocation the minimum fanout for allocation is increased by 2000.

These settings for this allocation can be modified by adding the following commands in the additional command file declared in the **Advanced** → **Build System Parameters** frame:

```
synchro fanout_min -value=<min_value>  
synchro fanout_step -value=<step_value>
```

Where <min_value> and <step_value> are integers.

Example:

If the highest fanout clocks in the design have fanouts of 4000 (clk1) , 3000 (clk2) and 2000 (clk3), the following BUFG are allocated when the default values for fanout_min and fanout_step are used:

- 1 BUFG for clk1 (highest clock with fanout ≥ 0)
- 1 BUFG for clk2 (fanout ≥ 2000)
- clk3 will be mapped to standard FPGA resources because its fanout is lower than 4000.

If the values are modified to reduce the step:

```
synchro fanout_step -value=1000
```

- 1 BUFG for clk1 (highest clock with fanout ≥ 0)
- 1 BUFG for clk2 (fanout ≥ 1000)
- 1 BUFG for clk3 (fanout ≥ 2000)



5.3 Compiling to Optimize Runtime Performance

The options and features described in this section should be selected for optimization purposes only, once the design is already up-and-running.

5.3.1 Timing-driven Optimization

The ZeBu compiler supports optimization of the most critical nets of the design in order to achieve a higher runtime performance.

This timing-driven optimization is available by selecting the **Use Timing-driven Mode** checkbox in the **Advanced** panel (**System Place & Route Parameters** frame).

5.3.2 Reduction of Inter-FPGA Clocks

It is possible to reduce the number of inter-FPGA clocks in order to reduce the `zClockSkewOffset` value and thus increase the maximum achievable runtime frequency. For that purpose, the appropriate clock trees are replicated in order to remove all inter-FPGA clocks when the **Single FPGA clock path localization** checkbox is selected in the **Clock Handling** panel.

When this feature is selected, the cut of the design FPGAs may be slightly higher but with no significant impact on the overall performance of the emulation (compared to the gain of reducing the clock skew).

However, the FPGA compilation may be longer because of higher filling rates due to the replicated clock trees. This feature also impacts the debugging capability because the signals in the replicated clock trees appear several times in the runtime database and make the CSA feature impossible to use.

5.4 FPGA Place and Route

5.4.1 Introduction

The ZeBu delivery package includes a specific Xilinx ISE package for ZeBu. It is a subset of Xilinx ISE which supports FPGA Place and Route for ZeBu, as described in the [ZeBu Installation Manual](#).

The version of the Xilinx ISE place and route software which has been tested for a given ZeBu software version is mentioned in the corresponding [ZeBu Release Note](#).

5.4.2 Parallel Automatic Recompilation of Failing FGAs (PARFF)

For designs with a high number of FGAs and when the FPGA filling rates are higher than the clustering default settings, the FPGA Place & Route of the design sometimes failed for very few FGAs which would need different settings. In order to take such FGAs into account, the ZeBu compiler can automatically launch the FPGA Place & Route with different settings if it fails with the default settings or if map step in ISE is very long.

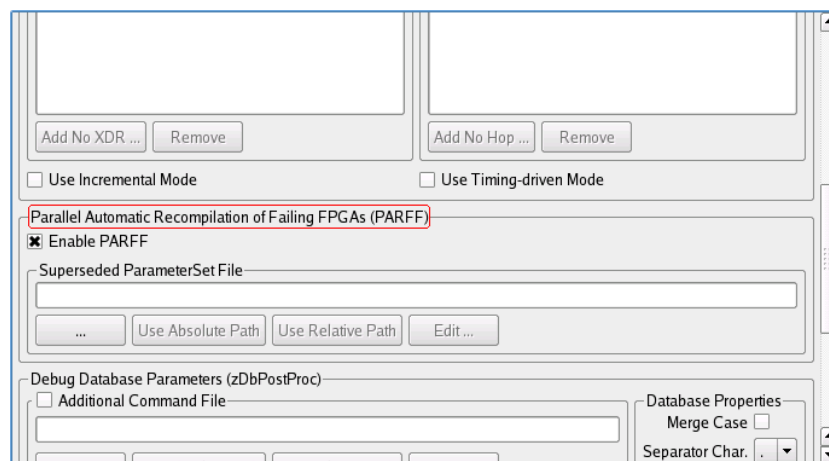
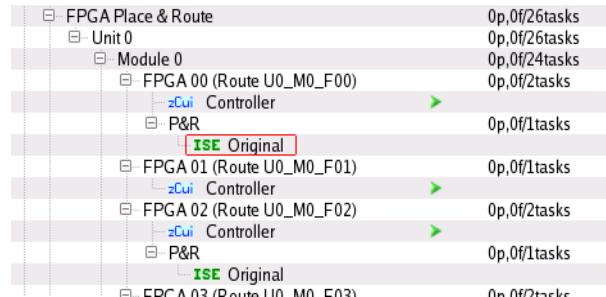


Figure 59: Advanced → PARFF frame

To use the PARFF feature, it is strongly recommended to use a compilation farm and declare remote commands for FPGA P&R as described in Section 4.1.3 and configure the retry mechanism as described in Section 4.1.4.

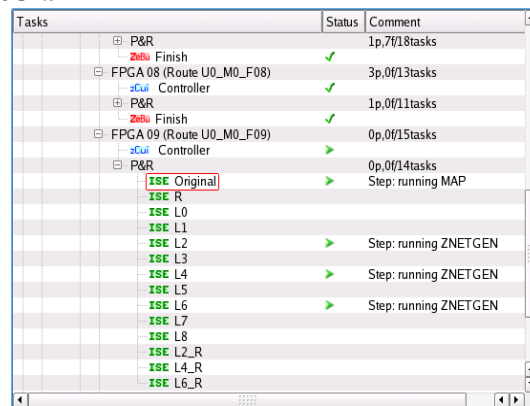
When the **Enable PARFF** option is selected in the **Advanced** panel (**Parallel Automatic Recompilation of Failing FPGAs** frame), the ZeBu compiler first launches FPGA P&R with the default settings, which is similar to FPGA P&R without PARFF activated. It is displayed as **Original** in the FPGA Place & Route task group of the **Compilation** view:



FPGA Place & Route	0p,0f/26tasks
Unit 0	0p,0f/26tasks
Module 0	0p,0f/24tasks
FPGA 00 (Route U0_M0_F00)	0p,0f/2tasks
zCui Controller	0p,0f/1tasks
ISE Original	0p,0f/1tasks
FPGA 01 (Route U0_M0_F01)	0p,0f/1tasks
zCui Controller	0p,0f/1tasks
FPGA 02 (Route U0_M0_F02)	0p,0f/2tasks
zCui Controller	0p,0f/1tasks
ISE Original	0p,0f/1tasks

Figure 60: *Original* FPGA P&R task

Some criteria have been defined so that the ZeBu compiler automatically launches several FPGA Place & Route tasks in parallel with different set of parameters before finishing the **Original** task:




Tasks	Status	Comment
P&R		1p,7f/18tasks
Finish	✓	3p,0f/13tasks
FPGA 08 (Route U0_M0_F08)	✓	1p,0f/11tasks
zCui Controller	✓	0p,0f/15tasks
P&R	➤	0p,0f/14tasks
ISE Original	➤	Step: running MAP
ISE R	➤	Step: running ZNETGEN
ISE L0	➤	Step: running ZNETGEN
ISE L1	➤	Step: running ZNETGEN
ISE L2	➤	Step: running ZNETGEN
ISE L3	➤	Step: running ZNETGEN
ISE L4	➤	Step: running ZNETGEN
ISE L5	➤	Step: running ZNETGEN
ISE L6	➤	Step: running ZNETGEN
ISE L7	➤	Step: running ZNETGEN
ISE L8	➤	Step: running ZNETGEN
ISE L2_R	➤	Step: running ZNETGEN
ISE L4_R	➤	Step: running ZNETGEN
ISE L6_R	➤	Step: running ZNETGEN

Figure 61: Task tree showing relaunched and retried P&R tasks

If the CPU load is too high for a given FPGA P&R task, the same set of parameters is used to launch the task on a different PC of the compilation farm. The new task can be tracked in the task tree because it is named with an “R” (**R** is a retry task for **Original** and **L2_R** is a retry task for **L2** set of parameters).

When one of the FPGA P&R tasks terminates successfully, all the running tasks for this FPGA are aborted and the pending tasks are cancelled (the intermediate files of aborted and cancelled tasks are automatically deleted). The **P&R** item of the task tree is automatically collapsed and the **Finish** task is shown with the ✓ status icon.

Note that some issues with network file system may cause some inappropriate failed status for the **Finish** task; in such cases, the **File System Options** must be configured as described in Section 4.1.4.

When expanding the **P&R** item, the aborted FPGA P&R tasks are shown with the  status icon and the cancelled tasks are shown with the  status icon:

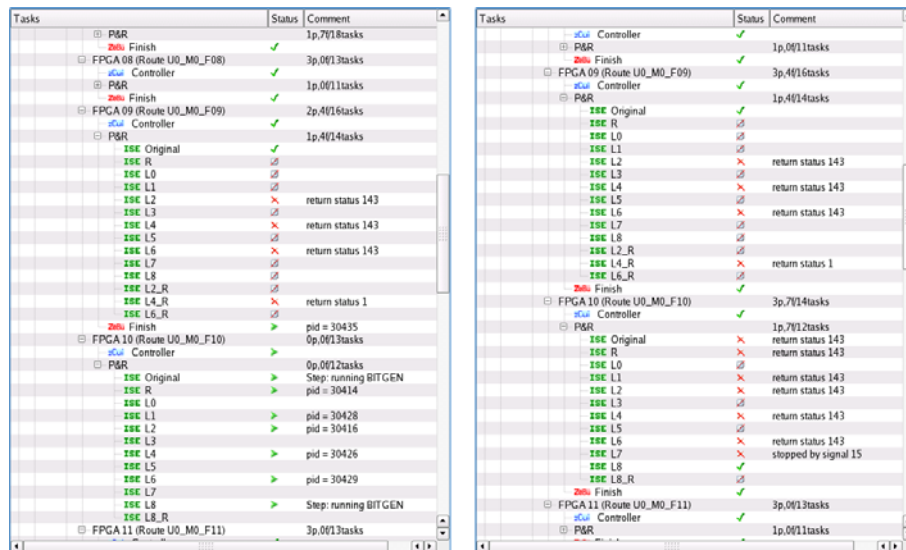


Figure 62: Task tree showing FPGA P&R tasks

If all the FPGA P&R tasks fail, all the intermediate files are kept in the compilation filetree for investigation.

The following table shows the specific options which are set for the different set of parameters:

Table 7: FPGA P&R Parameter Sets for PARFF

Task Id	Corresponding ISE parameters
L0	MAP_OPTIONS="-ol high -t 2"
L1	MAP_OPTIONS="-ol high -t 3"
L2	MAP_OPTIONS="-ol high -t 4"
L3	MAP_OPTIONS="-ol high -t 5"
L4	MAP_OPTIONS="-ol high -t 6"
L5	XIL_MAP_FULLPACK="1"
L6	XIL_PAR_ENABLE_LEGALIZER="1"
L7	UAP_CLIQUÉ_LIMIT="10" RT_NO_PLAYOPT="1"
L8	UAP_CLIQUÉ_LIMIT="10" RT_NO_PLAYOPT="1" XIL_PAR_ENABLE_LEGALIZER="1" XIL_PAR_CLKPH2_MAXIMIZE="1" XIL_MAP_FULLPACK="1"

If none of the predefined sets of parameters leads to a successful compilation, a user set of parameters can be declared in the **Superseded ParameterSet File** of the **PARFF** frame. The parameter file overwrites the L0 to L8 parameters given in Table 7.

5.4.3 Manual Modification of FPGA Place & Route Options

In addition to Parallel Automatic Recompilation of Failing FPGAs (PARFF), the FPGA Place and Route settings can be modified manually from the **zCui** graphical interface. The modifications in the **FPGA P&R** panel are possible after the first compilation has gone through FPGA Place and Route but they are applicable only for the **Original** FPGA P&R task:

- When PARFF is disabled, the FPGA P&R will be launched once for each FPGA with the user-defined settings.
- When PARFF is active, the original P&R task uses the user-defined settings but in case of failure, the predefined set of parameters is used as described in Section 5.4.2.

The **FPGA ISE Parameters** frame includes two areas:

- The left-hand area shows which options have been previously set for the FPGAs, either restored from the compilation project file (zpf file) or set for the current compilation.
- The right-hand area shows the FPGAs which are used for the design but which do not have modified P&R settings.

For both areas, some filters are available to display a limited number of FPGAs, as described in Section 5.4.3.3.

5.4.3.1 Before the first compilation

When no compilation has previously gone through FPGA P&R and if the loaded compilation project file did not include some FPGA P&R settings, the frame displays empty areas:

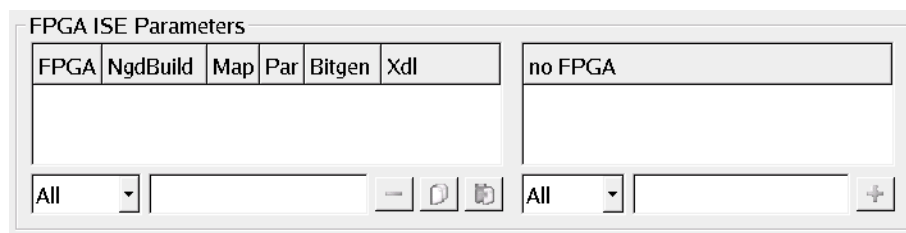


Figure 63: FPGA Parameters before any compilation

5.4.3.2 After Compilation

Once the compilation has gone through FPGA P&R for the first time, the right-hand area is updated with the list of FPGAs which were actually used by this compilation:

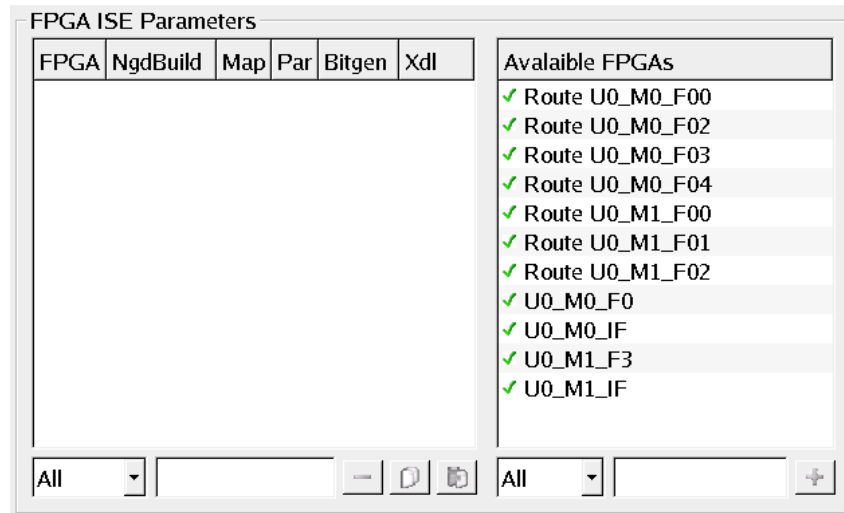


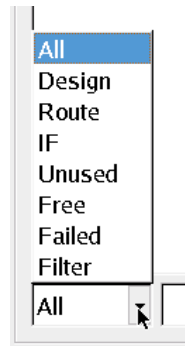
Figure 64: FPGA Parameters after first compilation

The FPGA listed in the right-hand area have a green tick (✓) if ISE passed successfully and a red cross (✗) if ISE failed. Note that the failed status is based on the availability of the binary file (.bit) for the FPGA.

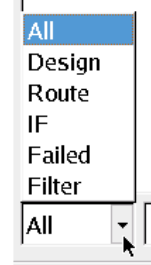
The left-hand area displays a table in which rows are FPGAs and columns are the tools of ISE FPGA Place and Route process (Ngdbuild, Map, Par, Bitgen and Xdl). To set an option, double-click on the cell for the FPGA and the tool and declare the option in the text edition field.

A death's head icon is displayed if the FPGA previously had specific options but is no longer used in the compilation (unused FPGA).

5.4.3.3 Description of the filters



Filter for left-hand area



Filter for right-hand area

Figure 65: Filters for FPGA ISE Parameters

For each area, the following filters are available:

- **All** Shows all FPGAs that are expected in the area.
- **Design** Shows only the FPGAs on which the design is mapped (FPGAs listed as **Ux_My_Fz**).
- **Routing** Shows only the FPGAs which are used for the communication infrastructure, without any design mapped (FPGAs listed as **Route Ux_My_Fz**).
- **IF** Shows only the interface FPGAs of all the modules of the Zebu-Server units (FPGAs listed as **Ux_My_IF**).
- **Unused** Shows only the FPGAs which are not used in the current compilation but for which options are set (FPGAs with death's head icon).
- **Free** Shows only FPGA for which no specific options are set.
- **Failed** Shows only the FPGAs which did not pass the FPGA P&R with their current options (FPGAs with red cross icon)
- **Filter** Shows only the FPGAs which match the content of the adjacent text input field.

Note that **Unused** and **Free** values are not applicable for the right-hand area.

5.4.3.4 Reusing an existing compilation project

When an existing compilation project is loaded in **zCui**, it may include some FPGA Place and Route settings from a previous compilation. In such a case, both areas of the FPGA settings frame display the corresponding status:

- The left-hand area lists the FPGAs for which options were stored in the **zCui** project file. A question mark icon is displayed in front of the FPGAs.
- The right-hand area is empty.



5.5 Information for Additional Command Files

The Zebu compiler supports various additional command files in which advanced commands can be added. Most of these files are Tcl compliant script; only the **Additional zFAST Attribute File** in the **zFAST** tab has a specific syntax, described in the ***zFAST Synthesizer Manual***.

5.5.1 Declaring commands for the zCore-level Compiler

The system-level compiler can transmit commands to the zCore-level compiler, either for all the zCores of the design or for a specific zCore. Such commands can be declared one-by-one or through dedicated Tcl script files.

5.5.1.1 Command-by-command Declaration

To declare a zCore-level command, the following line should be added in the **Build System** → **Advanced Command File (Advanced panel)**:

```
| zcorebuild_command <zcore_name> <zcore_command>
```

Where <zcore_name> is either the name of a zCore declared in the **zCore Definition File** or * to use the script for all the zCores of the design; <zcore_command> is a command for the zCore-level compiler.

If the command has some arguments and/or options (including spaces), the following syntax is applicable:

```
| zcorebuild_command <zcore_name> {<zcore_command with parameters>}
```

5.5.1.2 zCore-level Script Declaration

To declare a Tcl script for the zCore-level compiler, the following line should be added in the **Build System** → **Advanced Command File (Advanced panel)**:

```
| zcorebuild_script <zcore_name> <zcore_script.tcl>
```

Where <zcore_name> is either the name of a zCore declared in the **zCore Definition File** or * to use the script for all the zCores of the design; <zcore_script.tcl> is the script that is intended for the zCore-level compiler.

As for any other path declared in a command file, only absolute paths should be used for <zcore_script.tcl> because the actual origin of a relative path may change with the actual back-end target set in **zCui**.



5.5.2 Automatic zCore Generation

An additional command file can be declared in the **Automatic zCore Generation** frame of the **Clustering** panel, modify the parameters of the partitioning algorithm, in particular the recommended FPGA resources, or to declare instances and modules which should not be split in separate zCores.

The following commands are supported:

Command	Description
defcore_resource	Defines the constraints for a type of resources (registers, LUTs, BRAMs, DSP) that are assigned per zCore.
set_indivisible_instance	Sets an instance indivisible (it will not be splitted by the partitioner).
set_indivisible_module	Sets a module as indivisible (each instance of the given module will not be split by the partitioner)
set_max_blocks	Sets the max number of blocks of the design will load, default is 0 (automatically computed)
set_max_iter	Sets the number of iterations in the process. Default is 5. Reducing it provides the solution faster; increasing it ...
set_max_parts	Sets the maximum number of zCores that are created.

Example:

```
set_max_parts 5
set_max_iter 20
#set_max_blocks 10000

# Following values are the default ones
#defcore_resource -min LUT 3200000
#defcore_resource -min REG 1200000
#defcore_resource -min RAM 3500
#defcore_resource -min DSP 100

# Following values are the modified settings
defcore_resource -max LUT 3500000
defcore_resource -max REG 1400000
defcore_resource -max RAM 4000
defcore_resource -max DSP 500

# To keep dut.counter instance as a single block
set_indivisible_instance dut.counter
```



5.5.3 Advanced Commands for Performance Oriented Partitioning

An additional command file can be declared in the **Performance Oriented Partitioning for zCores** frame of the **Clustering** panel, to modify the parameters of the partitioning algorithm, in particular the recommended FPGA resources.

Command	Description
defgroup	Defines a group of instances described through their instantiation path.
defmapping	Defines an assignment of an instance described through its instantiation path to a target fpga.
disable	Disables an optional algorithm.
enable	Enables an optional algorithm.
flatten_instance	Flattens an instance before proceeding with partitioning
godown_instance	Forces the partitioner to go down the instance
lockfpga	Locks a FPGA so that no other blocks than pre-assigned blocks (are added to this FPGA.
set_filling_rate	Defines the FPGA filling rate constraints for a type of resources (registers, LUTs, BRAMs, DSPs).
set_indivisible_instance	Declares an instance that must not be split on several FPGAs by the partitioner
set_indivisible_module	Declares a module for which instances must not be split on several FPGAs by the partitioner
set_lut_weighting	Sets the LUT weighting.
set_max_blocks	Sets the max number of blocks of the design will load, default is 0 (automatically computed)
set_max_iter	Sets the maximum number of iterations of the partition.
set_max_time	Sets the maximum duration for an iteration of the partitioner (in minutes).
set_primitive_resource	Sets the resource cost for a given Xilinx primitive.
set_resource_critical	Sets a resource as critical.



6 ZeBu-Server Document Package

For each version, the *ZeBu Release Note* describes the new features, the fixed bugs, the known limits, the evolutions of the documentation package and the compatibility information.

The following Manuals constitute the ZeBu documentation package (some are generic manuals for the ZeBu range):

- [0] The *ZeBu-Server Installation Manual* describes how to install the ZeBu software and hardware.
- [1] The *ZeBu-Server Compilation Manual* describes the compilation process.
- [2] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.
- [3] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.
- [4] The *ZeBu zRun Emulation Interface Manual* describes the zRun emulation control interface and how to use the different functions.
- [5] The *ZeBu-Server Smart Z-ICE Manual* provides detailed information on how to configure and to connect the Smart Z-ICE interface to an external system.
- [6] The *ZeBu-Server Direct ICE Manual* provides detailed information on how to configure and to connect an ICE module for connection emulated DUT I/O pins to a target system and hard cores.
- [7] The *ZeBu C API Reference Manual* and *ZeBu C++ API Reference Manual* provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ test bench to verify your design.
- [8] The *ZEMI-3 Manual* introduces the ZEMI-3 infrastructure together with the advantages of transaction-based verification. It presents ZEMI-3 features and gives elements to choose the most appropriate architecture for your transactor with recommendations to write the HW and SW parts of your transactor.
- [9] The *zFAST Synthesizer Manual* describes the use of **zFAST**, the ZeBu-dedicated synthesizer, integrated in both standard mode and script mode in **zCui**. Advanced information is also available for **zFAST** attributes and for the **zFAST** Stat Browser.



7 EVE Contacts

For product support, contact: support@eve-team.com.

For general information, visit our company web-site: <http://www.eve-team.com>

Europe Headquarters	EVE SA 2bis, Voie La Cardon Parc Gutenberg, Batiment B 91120 PALAISEAU France Tel: +33-1-64 53 27 30
US Headquarters	EVE-USA, Inc. 2290 N. First Street, Suite 304 San Jose, CA 95131 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 4F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE DESIGN AUTOMATION Pvt. Ltd. # 143, First Floor, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala, Bangalore - 560 095 Karnataka, INDIA Tel: +91-80-41460680/ 30202343
Taiwan Headquarters	14F1, No 371, Sec. 1 Guangfu Rd., East District Hsinchu City 300 Taiwan (R.O.C.) Tel: +886 (3) 564 7900