



The Fastest Verification

ZeBu™ zFAST Synthesizer Manual

Document revision – a –

June 2010

Version 6_2_1

Copyright © 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 © 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	6
OVERVIEW	6
HISTORY	6
RELATED DOCUMENTATION	6
TYPOGRAPHIC CONVENTIONS USED IN THIS MANUAL	7
1 INTRODUCTION	8
2 PREREQUISITES	9
2.1 SUPPORTED LANGUAGES	9
2.2 LAUNCHING zCUI	9
2.3 CONFIGURATION OF THE COMPILATION PROJECT.....	10
2.3.1 <i>Filling the Project Properties and Design Properties panels</i>	10
2.3.2 <i>RTL Groups</i>	11
2.3.3 <i>Transactors</i>	11
2.3.4 <i>Preferences Panel</i>	12
3 zFAST STANDARD MODE.....	13
3.1 CHOOSING zFAST STANDARD MODE.....	13
3.2 IMPORTING THE SOURCE FILES.....	14
3.2.1 <i>Adding the Source Files to the Project</i>	14
3.2.2 <i>Declaring the Working Library for a source file</i>	16
3.3 CONFIGURING SYNTHESIS WITH zFAST	17
3.3.1 <i>Configuration items in zCui</i>	17
3.3.2 <i>Settings in the Main tab</i>	18
3.3.3 <i>Settings in the zFAST tab</i>	19
3.3.4 <i>Default Settings</i>	19
3.3.5 <i>Modifying the Settings</i>	20
4 zFAST SCRIPT MODE.....	21
4.1 INTRODUCTION	21
4.2 SPECIFIC FEATURES OF THE SCRIPT MODE	22
4.2.1 <i>Settings for the RTL Source Files</i>	22
4.2.2 <i>Elaboration Features</i>	22
4.3 CHOOSING zFAST SCRIPT MODE.....	23
4.4 WRITING THE zFAST SCRIPT	24
4.4.1 <i>General syntactical rules for the zFAST script</i>	24
4.4.2 <i>Declaring the Source Files for RTL Analysis</i>	25
4.4.3 <i>Elaboration</i>	27
4.4.4 <i>Specific Commands</i>	27
4.4.5 <i>Basic Example of zFAST Script</i>	28
4.4.6 <i>zFAST Script Example with Options</i>	29
4.4.7 <i>zFAST Script Example with Advanced Options for a Verilog design</i>	29
4.4.8 <i>List of Commands for the zFAST Script</i>	30



5	RUNNING AND MONITORING SYNTHESIS	31
5.1	RUNNING SYNTHESIS	31
5.2	INCREMENTALITY	32
5.3	MONITORING THE SYNTHESIS	32
5.4	zFAST OUTPUTS	33
5.5	ANALYZING THE zFAST LOG FILES	34
5.5.1	<i>Search Capability in zCui</i>	<i>34</i>
5.5.2	<i>Investigating Memories inferred with zMem</i>	<i>35</i>
5.5.3	<i>Analyzing the Post Processing Log File</i>	<i>35</i>
5.6	PROCEEDING WITH BACK-END COMPILATION	35
6	LANGUAGE-SPECIFIC FEATURES.....	36
6.1	VERILOG AND SYSTEMVERILOG SPECIFIC FEATURES.....	36
6.1.1	<i>Definition Rules for Blackboxes</i>	<i>36</i>
6.1.2	<i>pullup/pulldown Verilog Primitives</i>	<i>36</i>
6.1.3	<i>trio/tril/trireg wires</i>	<i>36</i>
6.1.4	<i>Verilog strength</i>	<i>36</i>
6.1.5	<i>pmos/nmos primitives</i>	<i>36</i>
6.1.6	<i>#delay statements</i>	<i>37</i>
6.1.7	<i>\$display & others Verilog tasks</i>	<i>37</i>
6.1.8	<i>Memory Initialization with \$readmemb/\$readmemh</i>	<i>37</i>
6.1.9	<i>Memory inference</i>	<i>37</i>
6.1.10	<i>Non-RTL language support</i>	<i>40</i>
6.2	SYNTHESIS PRAGMAS	40
7	WRITING AN ATTRIBUTE FILE FOR ZFAST	42
7.1	SYNTAX RULES	42
7.2	LIST OF ATTRIBUTES	43
8	ADVANCED USAGE	45
8.1	RTL-BASED COMPILATION SCRIPTS	45
8.1.1	<i>Introduction</i>	<i>45</i>
8.1.2	<i>Limitation for Escaped RTL Identifiers</i>	<i>46</i>
8.1.3	<i>Support of generate blocks</i>	<i>48</i>
8.2	CHUNKING	48
8.3	USING zFAST STAT BROWSER	49
8.3.1	<i>Launching the zFAST Stat Browser from zCui</i>	<i>49</i>
8.3.2	<i>Available Statistics for the design</i>	<i>50</i>
8.3.3	<i>Browsing the design to get statistics</i>	<i>51</i>
8.3.4	<i>Statistics Spreadsheets</i>	<i>52</i>
8.3.5	<i>Additional Features</i>	<i>54</i>
9	LIMITATIONS	55
10	ZEBU-SERVER DOCUMENTS	56
11	EVE CONTACTS	57



Figures

Figure 1: <i>Project Properties</i> panel in zCui	10
Figure 2: <i>Design Properties</i> panel in zCui	10
Figure 3: <i>Project Job Scheduler Preferences</i> panel in zCui	12
Figure 4: Selecting zFAST Standard Mode	13
Figure 5: Contextual menu of an RTL group	14
Figure 6: Contextual menu of an RTL source file	14
Figure 7: <i>RTL Group Properties</i> panel with RTL source files added.....	15
Figure 8: Working Library Declaration for an RTL Source File	16
Figure 9: Working Library displayed in zCui	16
Figure 10: <i>RTL Group Properties</i> panel with all tabs visible	17
Figure 11: <i>RTL Group Properties</i> panel with scrolling arrows.....	17
Figure 12: zFAST tab	19
Figure 13: Selecting zFAST Script Mode.....	21
Figure 14: <i>Properties</i> panel for zFAST script mode	23
Figure 15: <i>Alias</i> tab for zFAST script mode	23
Figure 16: Adding a Script File in zCui	24
Figure 17: Running zCui for Synthesis only.....	31
Figure 18: Selecting the compilation target for synthesis.....	31
Figure 19: Synthesis steps in <i>Compilation View</i>	33
Figure 20: Accessing the zFAST Stat Browser from zCui	49
Figure 21: zFAST Stat Browser Opening Window	50
Figure 22: zFAST Stat Browser <i>Main Window</i>	51
Figure 23: zFAST Stat Browser <i>Main Window</i> with expanded hierarchy	51
Figure 24: zFAST Stat Browser <i>Details Window</i>	52
Figure 25: zFAST Stat Browser - Statistics Spreadsheet for Modules.....	52
Figure 26: zFAST Stat Browser - Statistics Spreadsheet for Memories.....	53
Figure 27: zFAST Stat Browser – <i>Tools</i> → <i>Properties</i>	53
Figure 28: zFAST Stat Browser – Sorted-out Statistics Spreadsheet.....	53

Tables

Table 1: Options for Verilog analysis command.....	25
Table 2: Options for VHDL analysis command.....	27
Table 3: List of Commands for zFAST Script	30
Table 4: zFAST Attributes to optimize design memories.....	39
Table 5: Syntax Rules for zFAST Attribute File	42
Table 6: List of zFAST Attributes	43



About This Manual

Overview

This 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.

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
a	6_2_1	June '10	First Edition

Related Documentation

The complete ZeBu-Server documentation package is listed in Chapter 10 of this manual.

The following documents provide additional information related to the present manual:

- The *[ZeBu-Server Compilation Manual](#)* provides general information about **zCui** and details the compilation process. Basic information is available for synthesis (standard mode only) and the back-end compilation is described with details.
- The following *[ZeBu Application Notes](#)* provides **zFAST**-related information:
 - SystemVerilog Assertions for ZeBu (AN028).
 - zDPI Feature for ZeBu (AN029).



Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space “Courier New” font, e.g. **zBuild**.

Input file contents, such as code examples, scripts or driver declarations are shown in mono-space “Courier New” font with left and bottom borders.

For example, the following line of a script describes a memory command that accepts a user-declared variable and a user-selected option:

```
memory set_rw_mode <port-name> [rw-mode]
```

Where:

<port-name> User-declared variables are inside angled brackets (“<” and “>”), such as a port name in the present example. Do not type angled brackets when changing parameters to user names.

[rw-mode] Options are presented inside square brackets. The available options will be listed and described in the text following the syntax rules. Do not type square brackets when changing options to user names.

Example:

```
memory set_rw_mode myPort readbeforewrite
```

Shell command lines are shown in mono-space “Courier New” font with left and bottom borders, including the shell prompt (which depends on your configuration); the command itself may be shown in bold characters for easier reading. :

```
$ zBuild <script_file_name>
```

Parameters and options are displayed in the same way as for input files contents: parameters inside angles brackets and options inside square brackets.

The shell prompt in this manual is \$ for user environment and # for supervisor environment.

Reports, logs and any output data (except Tcl scripts), generated by ZeBu tools are shown in mono-space “Courier New” font with complete border. The following example is a log header:

```
#####  
# Copyright (c) 2002-2005                               #  
# Emulation and Verification Engineering SA               #  
#-----#  
# <Tool Name>                                           #  
# revision :                                           #  
# date :                                           #  
#-----#  
#####
```

GUI elements (menu items, buttons, check boxes, edition fields) are shown in bold characters. Following is an example of GUI description:

In the **Generate** menu, select the appropriate wrapper type to generate for the probes and then select **Generate**.



1 Introduction

The ZeBu compilation flow includes a proprietary synthesizer named ZeBu Fast Synthesis (**zFAST**). **zFAST** enables faster synthesis than most commercial synthesis tools and provides full design accessibility for easier debug at runtime.

zFAST is fully integrated in **zCui** in addition to the existing support of commercial synthesizers. **zCui** provides an easy to use push-button interface with a use model similar to commercial synthesis tools.

zFAST provides the following key features:

- Support of Verilog, SystemVerilog, VHDL, and mixed HDL designs.
- Block-Based or Top-Down synthesis.
- Support of User Defined Primitives (UDP).
- Synthesis for RTL accessibility at runtime.
- Memory modeling capability for register-based, LUTRAM-based, BRAM-based and zrm-based memories with easy runtime access for any type of memory. The ZeBu Memory Generator (**zMem**) is automatically invoked in the **zFAST** synthesis flow if necessary.
- Support of SystemVerilog Assertions (SVA).
- Support of zDPI feature for DPI calls with input only in the SystemVerilog source files.
- Synthesis of transactors, in both zcei and ZEMI-3 environments.

Two synthesis modes are available with **zFAST**:

- The standard mode: all the RTL files and synthesis parameters are selected through **zCui**. This mode is described in Chapter 3.
- The script mode: a script similar to a simulation script is written for the synthesizer with the analysis and elaboration commands. This mode is described in Chapter 4.

In addition to **zCui** integration, **zFAST** synthesizer can be configured with additional attributes which are declared in a separate file, as described in Chapter 7, or in the **zFAST** script.

The **zFAST** resulting netlists can be directly used by the ZeBu back-end compilation tools.

Note that **zFAST** synthesizer is reserved for use with ZeBu and cannot be used in any other environment.



2 Prerequisites

This chapter provides recommendations which are applicable when synthesizing with **zFAST** standard mode or **zFAST** script mode.

2.1 Supported Languages

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 with SystemVerilog Assertions (SVAs).

In some cases, the language of the RTL source files can be detected automatically:

- When source files are declared as Verilog, **zFAST** can automatically distinguish between Verilog and SystemVerilog files.
- When source files are declared as SystemVerilog, **zFAST** does not automatically distinguish between Verilog and SystemVerilog files and Verilog files are synthesized as SytemVerilog files.
- **zFAST** does not automatically figure out the type for VHDL files.

2.2 Launching zCui

As stated in the *[ZeBu-Server Compilation Manual](#)*, 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 any pre-existing simulation environment.

It is also recommended to store the ZeBu-dedicated files (DVE file, memory description files, probe 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 the above 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:

- The following command line launches **zCui** for the creation of a new project:

```
$ zCui
```
- Once the compilation project (`.zpf` file) exists, you can launch **zCui** with the following command line:

```
$ zCui -p <my_project>.zpf
```

2.3 Configuration of the Compilation Project

2.3.1 Filling the *Project Properties* and *Design Properties* panels

When creating the compilation project in **zCui**, the **Project Properties** panel lists the environment of the compilation and provides a file selector for the **zCui** working directory (by default, the working directory is `./zcui.work`):

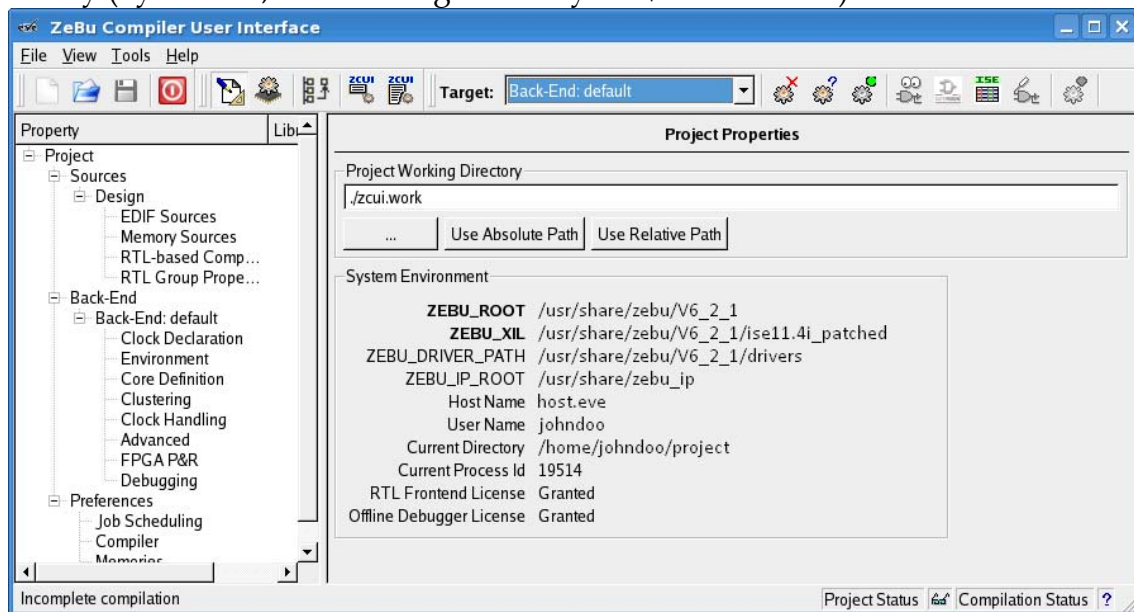


Figure 1: *Project Properties* panel in zCui

The **Design Properties** panel provides a text input field to declare the top of the design:

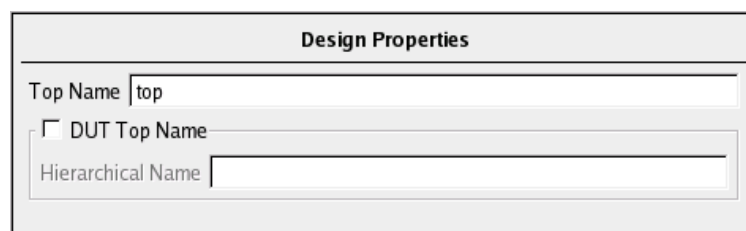


Figure 2: *Design Properties* panel in zCui

When the DUT is not the actual top level of the source files, in particular in case of a synthesizable testbench, select the **DUT Top Name** checkbox and add the path to the actual DUT top level in the corresponding text input field.



2.3.2 RTL Groups

In **zCui**, the RTL source files are gathered in one or several RTL Groups. Each RTL group has its own synthesis options. The synthesis output files of all RTL groups are automatically merged by the ZeBu compiler. By default, **zCui** creates one RTL group (**Default_RTL_Group**).

It is recommended to have only one RTL group for the project since most of the ZeBu debugging features are supported for one group only, in particular SystemVerilog Assertions, probe declaration with RTL names and CSA feature (Combinational Simulation for Accessibility). Note that hierarchical references in the RTL sources are supported only inside an RTL group (no hierarchical references between different RTL groups can be synthesized).

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

2.3.3 Transactors

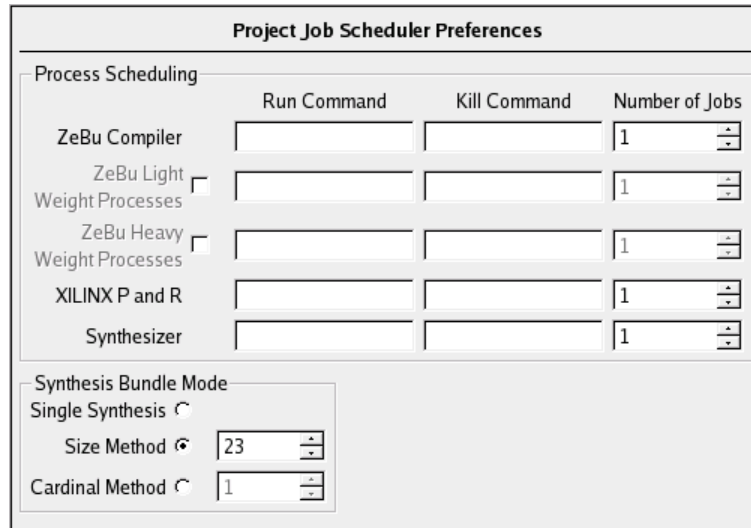
When synthesizing transactors for ZeBu, specific design units are created in the **Project Tree** pane (choose **Add ZEMI-3 Transactor...** or **Add Transactor...** from the contextual menu of **Project**, **Sources** or **Design** items).

Both ZEMI-3 and zcei-based transactors can be synthesized with **zFAST** in standard mode or script mode with the following limitations:

- Only one RTL group should be used, as described in Section 2.3.1.
- SystemVerilog Assertions (SVAs) and zDPI features are not supported.
- Hierarchical References can only be used within the transactor (no hierarchical reference to the DUT).
- The Combinational Signals Accessibility (CSA) feature is not supported.

2.3.4 Preferences Panel

In the **Preferences** panel, the size of the synthesis bundles can be configured:



Project Job Scheduler Preferences			
Process Scheduling			
	Run Command	Kill Command	Number of Jobs
ZeBu Compiler			1
ZeBu Light Weight Processes <input type="checkbox"/>			1
ZeBu Heavy Weight Processes <input type="checkbox"/>			1
XILINX P and R			1
Synthesizer			1

Synthesis Bundle Mode	
Single Synthesis <input type="radio"/>	
Size Method <input checked="" type="radio"/>	23
Cardinal Method <input type="radio"/>	1

Figure 3: Project Job Scheduler Preferences panel in zCui

A bundle gathers synthesis tasks that will be synthesized in a single process. Synthesis tasks are different depending on the synthesis mode:

- In top-down synthesis, only 1 synthesis task is created for the entire RTL group except if the chunking capability is activated in the **RTL Group Properties** → **zFAST** tab, as described in Section 8.2.
- In block-based synthesis, 1 synthesis task is created for each design unit (module/entity).

In the **Synthesis Bundle Mode** frame, the following choices are available for the compilation project (i.e. applicable for all RTL groups in the design and for all transactors):

- **Single synthesis:** **zCui** launches 1 synthesis process for each synthesis task.
- **Size Method:** **zCui** launches the appropriate number of synthesis processes to have in each bundle the average number of synthesis tasks selected in the corresponding spinbox.
- **Cardinal Method:** **zCui** launches as many synthesis processes as selected in the corresponding spinbox.

Note that the synthesis processes can be parallelized if a remote command has been specified in the **Process Scheduling** → **Run Command** → **Synthesizer** field and if the value for **Synthesizer** → **Number of Jobs** is higher than 1.

3 zFAST standard mode

This chapter describes how to use **zFAST** synthesizer standard mode with **zCui**.

3.1 Choosing zFAST standard mode

In the **RTL Group Properties** panel, the **zFAST** standard mode can be selected by choosing **zFAST** in the **Synthesizer** dropdown list:

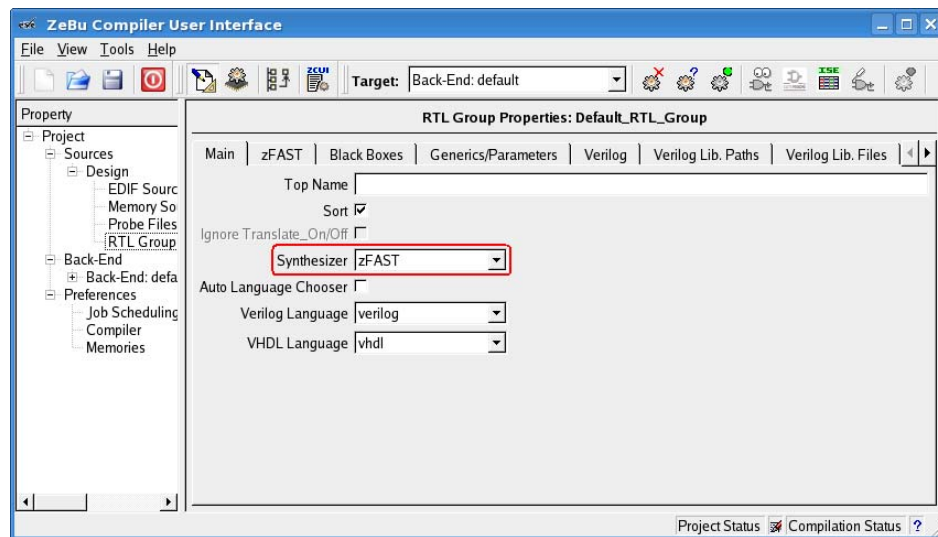


Figure 4: Selecting zFAST Standard Mode

In most cases, there is only one RTL group for the design and the choice of synthesizer has to be done for this group. If the verification environment includes user-written transactors, the choice of synthesizer has to be done for each transactor as well.

3.2 Importing the Source Files

3.2.1 Adding the Source Files to the Project

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** pane or on an existing RTL source file of the RTL group:

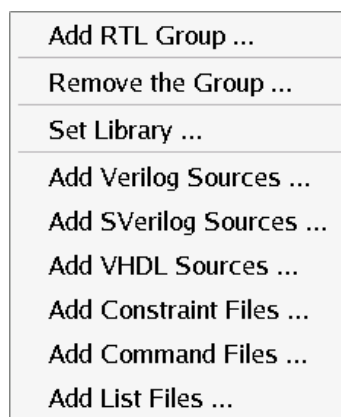


Figure 5:
Contextual menu of an RTL group

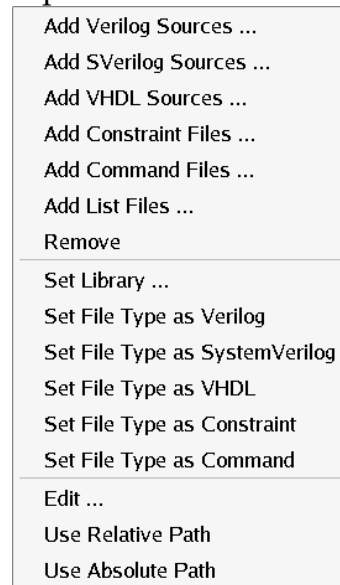





Figure 6: Contextual menu
of an RTL source file

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 (.f) 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 not relevant for **zFAST** standard mode.

An icon in front of the file name in the **Project Tree** pane shows the type of the source file:  for Verilog,  for SystemVerilog and  for VHDL.

In the contextual menu of an RTL source file (see Figure 6), some additional items are available:

- If an inappropriate file has been added, you can remove it with the **Remove** item in its contextual menu (Figure 6).
- If the type of a source file (Verilog, SystemVerilog or VHDL) is not correct, you can change it with the **Set File Type as...** item in the contextual menu of the file (Figure 6). The icon in front of the file name which shows the type of the file is automatically changed in such a case.
- 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 6). The editor can be changed in the **Preferences** → **Compiler** panel, as described in the *ZeBu-Server Compilation Manual*.

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

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

The following figure shows the **Project** view when RTL files have been added (2 RTL groups with Verilog files in this project):

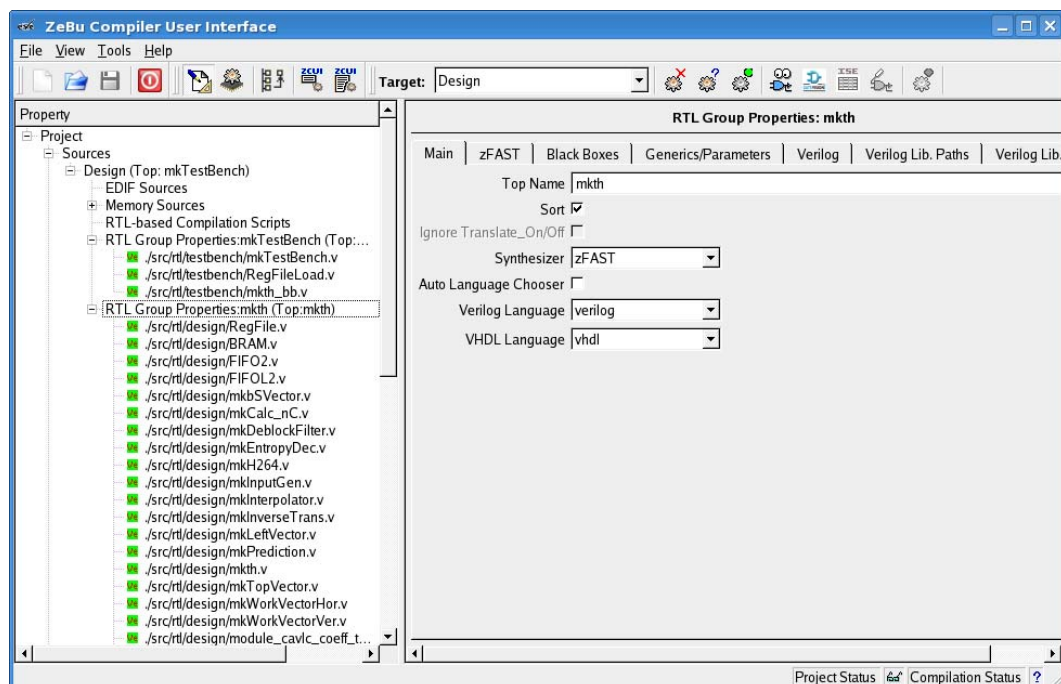


Figure 7: RTL Group Properties panel with RTL source files added

3.2.2 Declaring the Working Library for a source file

By default, the design is synthesized in the work working library.

This default working library can be declared for the entire RTL group and it can be modified on a file-by-file basis. When selecting **Set Library ...** in the contextual menu of the RTL group (see Figure 5) or of a given RTL file (see Figure 6), the same pop-up is displayed:

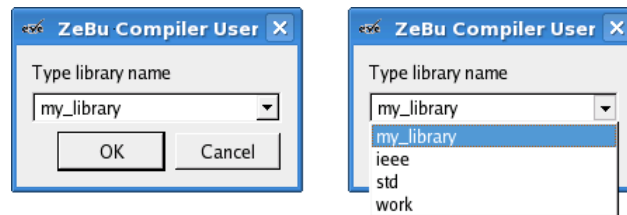


Figure 8: Working Library Declaration for an RTL Source File

To declare a working library which is not listed, `my_library` item can be modified manually and the corresponding library will be used for synthesis. If a specific library is declared by user, it is shown in the **Library** column of the **Project Tree** pane, as shown in the following example (`work` library has been declared for the entire RTL group and `RegFile.v` library has been set individually to `my_work_lib`):

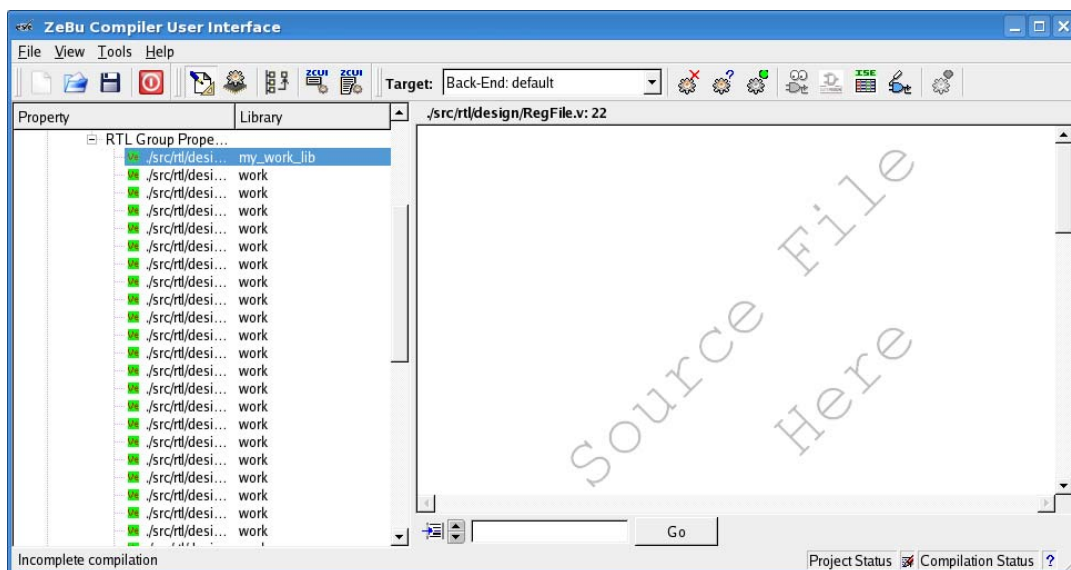


Figure 9: Working Library displayed in zCui

3.3 Configuring synthesis with zFAST

3.3.1 Configuration items in zCui

For each RTL group, the **RTL Group Properties** panel looks like the following example when **zFAST** is the selected synthesizer:

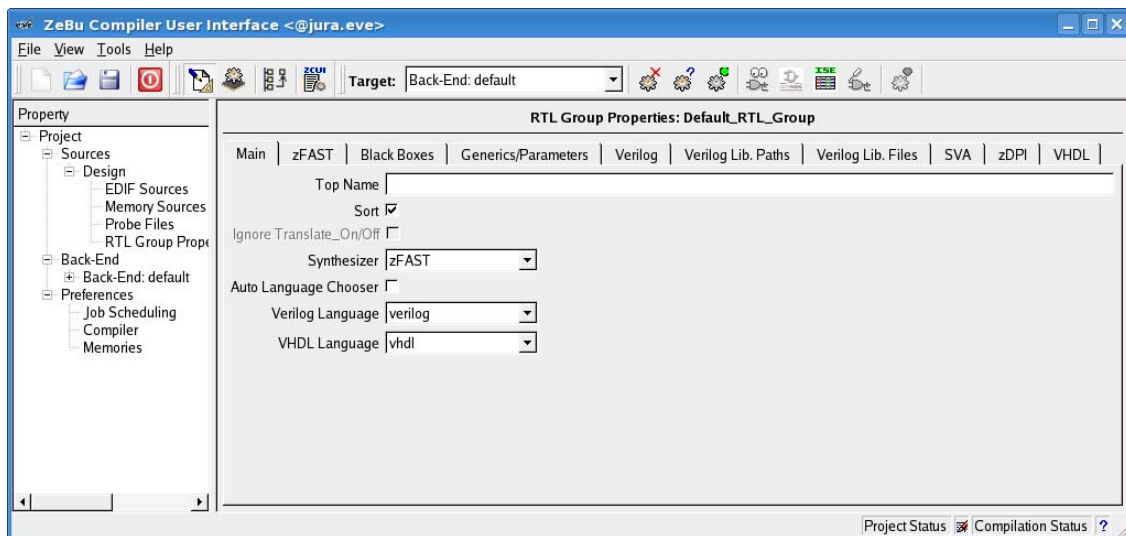


Figure 10: RTL Group Properties panel with all tabs visible

When the **zCui** display is not wide enough, in particular when it shows both the **Project Tree** pane and the **RTL Group Properties**, there may not be enough space to show all the tabs. In such a case, scrolling arrows become available in the top-right corner of the **Properties** panel:

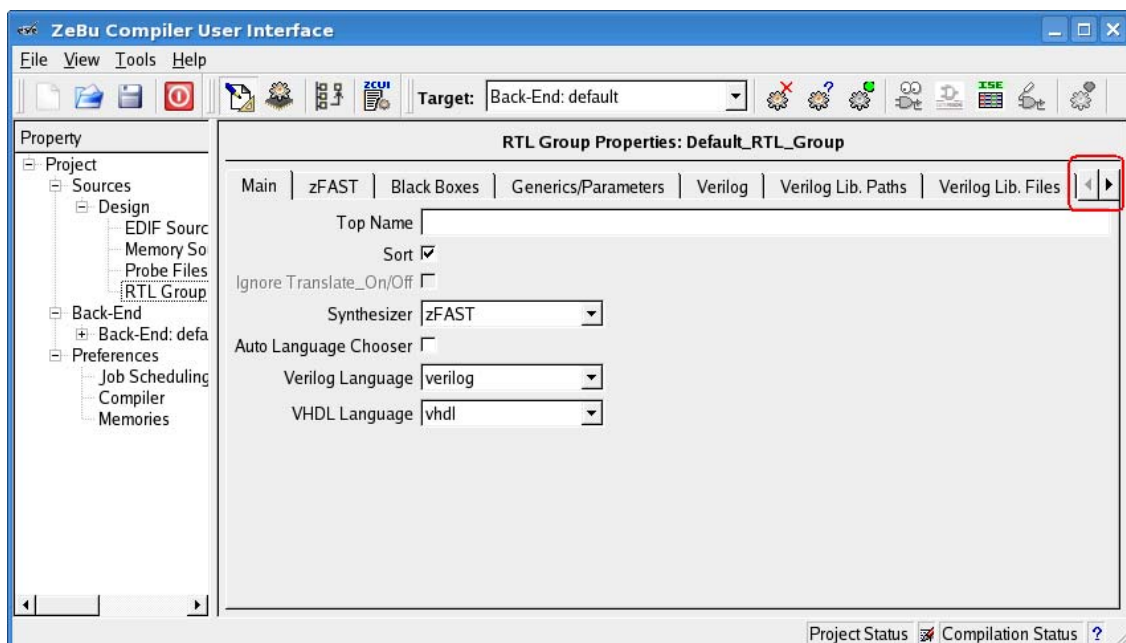


Figure 11: RTL Group Properties panel with scrolling arrows



3.3.2 Settings in the *Main* tab

None of the items of this tab are mandatory:

- 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 (see Section 2.3.1).
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 out 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, as described in Section 2.1. The file type selection is done on the file extension. This option is cleared by default.
- The **Verilog Language** and **VHDL Language** drop-down lists 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.
Note that when selecting **Verilog95** in the **Verilog Language** list, **zFAST** actually synthesizes the design with Verilog-2001 constraints, in particular usage of Verilog-2001 keyword in Verilog-95 code will cause synthesis errors.
 - VHDL-93 is the default for VHDL files. The other possible choice is VHDL-87 if the source files are not compatible with VHDL-93.

3.3.3 Settings in the zFAST tab

The **zFAST** tab includes specific options for **zFAST**, in particular the choice between block-based and top-down synthesis (**Synthesis Mode** option buttons) and the settings for optimization and debugging capability.

The **zFAST** tab looks like the following example:

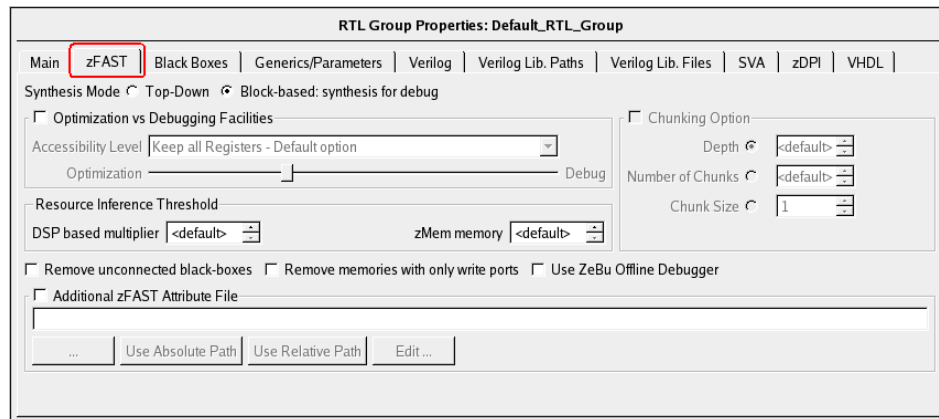


Figure 12: zFAST tab

When **Top-Down** mode is selected, the synthesis can be processed separately with chunking criteria, as described in Section 8.2.

For advanced configuration of the **zFAST** synthesizer, optional attributes can be set in a dedicated file which is declared for the entire RTL group in the **Additional zFAST Attribute File**. Detailed information about the **zFAST** attributes is available in Chapter 7.

3.3.4 Default Settings

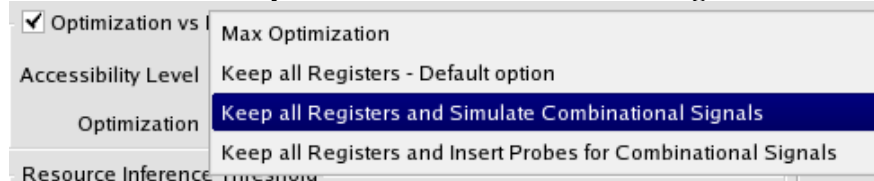
Following is the list of default settings for **zFAST** in **zCui** environment:

- Memories are automatically inferred with registers when smaller than 2,048 bits (overall size of the memory); bigger memories use **zMem** memory models.
- Multipliers are inferred with DSPs if the total number of bits of the two arguments is higher than 16 (port A size + port B size \geq 16 bits); smaller multipliers use FPGA LUTs.
- All the registers of the design are kept for accessibility through synthesis, but not combinational signals which can be simplified by **zFAST**. A selector in **zCui** modifies this setting, as described in Section 3.3.5.1.
- The default priority order is Read-After-Write for the **zMem** memory models created through **zFAST**.

3.3.5 Modifying the Settings

3.3.5.1 Modifying the Accessibility Selector

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



The choices in the **Accessibility Level** drop-down list are the following ones:

- **Max Optimization:** some registers may not be accessible at runtime; the synthesizer and the back-end compiler perform the best optimization.
- **Keep all Registers – Default option:** all registers are accessible. It will prevent them from being optimized by the synthesizer and the back-end compiler. Only the read access is guaranteed: the write access may not be possible if the register is driven by a constant or if it is mapped onto a non-writeable FPGA (FPGAs with RLDRAM memories in 4C and 8C/ICE modules). Combinational signals are not accessible, except if dynamic probes are inserted manually in an **RTL-based Compilation Script** file.

- **Keep All Registers and Simulate Combinational Signals:** all signals are accessible; the values of combinational signals will be simulated from register values.

In this mode, all the registers are kept through the compilation process and the combinational signals are available at runtime thanks to CSA feature (Combinational Signals Accessibility) without any post-compilation operation. Warning: The gate count of netlist reported by the **Post Process Front-End Results** step may be wrong due to the netlist instrumentation. The unused extra logic that might be inserted will be removed during the back-end compilation.

- **Keep All Registers and Insert Probes for Combinational Signals:** dynamic probes are added so that all registers and combinational signals can be accessed at runtime. However, this option drastically increases the actual logic size of the design.

4 zFAST Script mode

4.1 Introduction

In addition to the standard mode described in Chapter 3, **zFAST** provides a script mode which 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. Synthesis itself is not launched by the script: **zCui** launches the synthesis tasks, taking into account user settings in the graphical interface for chunks and bundles.

The script mode provides the same basic features as the standard mode since it is based on the same synthesizer and the synthesis output files are the same. As for standard mode, only one RTL group is recommended, as described in Section 2.3.1.

The script mode provides better flexibility than the standard mode for the source file-based options for synthesis:

- In standard mode, you can define which library is to be used for analysis of a given RTL source file but other synthesis options are not available on a file-by-file basis (only group-based options).
- In script mode, different options can be declared for each RTL source file in a Tcl script written by user: the **zFAST** script.

This chapter mostly describes the specific features of the script mode in comparison to the standard mode.

In **zCui**, the script mode is selected as a specific synthesizer in the **Properties** panel of the RTL group:

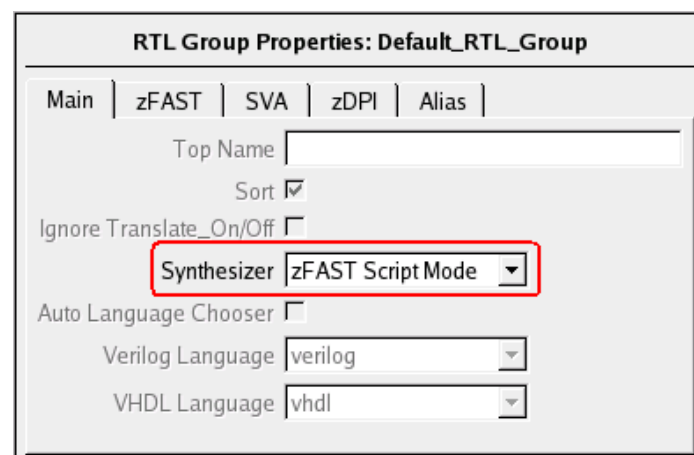


Figure 13: Selecting zFAST Script Mode

The tabs that are replaced by commands in the **zFAST** script are not available for the script mode.

All the other **zCui** features are unchanged: **zCui** manages the whole compilation process, including incremental compilation and job scheduling.



4.2 Specific Features of the Script Mode

4.2.1 Settings for the RTL Source Files

With the script mode, the declaration of the RTL source files is done by adding the command for RTL analysis in the script.

For Verilog and VHDL source files, the following information can be set in the script for each RTL source file:

- Library for RTL analysis.
- Language version or declaration of the file extensions for automatic detection of Verilog and SystemVerilog source files.
- Synthesis pragmas to be taken into account or ignored.
- Additional **zFAST** attributes applicable on a command-by-command basis (in addition to the **Additional zFAST Attribute File** which is applicable to the entire RTL group).

For Verilog source files only, the following information can also be declared in the script on a file-by-file basis:

- Include paths
- Macros
- Library paths (-y option)
- Library extensions
- Library files (-v option)
- Activation of library rescan

4.2.2 Elaboration Features

The script mode provides the same settings as the standard mode to configure the elaboration process:

- The name of the top-level module and associated parameters.
- Declaration of modules synthesized as blackboxes (similar to the **Black Boxes** tab in standard mode)
- For a VHDL design, declaration of the library, of the top-level architecture and of the configuration (if applicable)

4.3 Choosing zFAST Script Mode

When the script mode is selected, **zCui** shows the following tabs for the **Properties** panel:

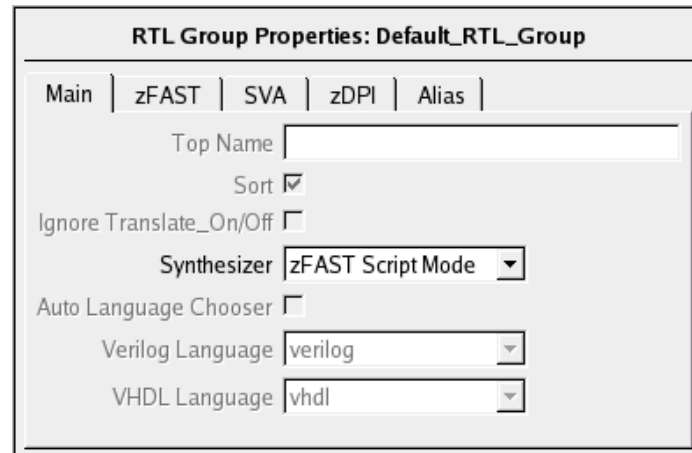



Figure 14: Properties panel for zFAST script mode

Only 5 tabs are available for when the script mode is selected:

- The tabs that are replaced by commands in the **zFAST** script are not available: **Black Boxes**, **Generics / Parameters**, **Verilog**, **Verilog Lib. Paths**, **Verilog Lib. Files** and **VHDL**.
- In the **Main** tab, all the items (except the choice of the synthesizer) are grayed out because they are replaced by commands in the **zFAST** script.
- The **zFAST**, **SVA** and **zDPI** tabs are unchanged with respect to the standard mode.
- The **Alias** tab is specific to the script mode: it is a subset of the **VHDL** tab of the standard mode and its content is applicable for both VHDL and Verilog languages:



Library Aliases:	
Alias	Target

Figure 15: Alias tab for zFAST script mode

Adding a **zFAST** script for the RTL Group is similar to adding a RTL source file in the standard mode: right-click on **RTL Group Properties** in the **Project Tree** pane and select the **Add Command Files...** item:

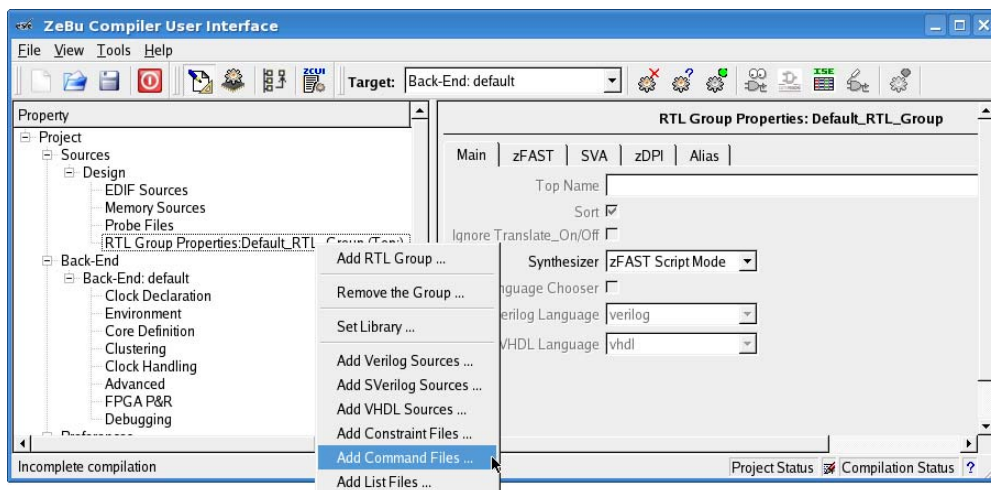


Figure 16: Adding a Script File in zCui

4.4 Writing the zFAST Script

The **zFAST** script is a Tcl script which implements a set of proprietary commands for the declaration of source files, for elaboration and to configure the synthesis process.

The complete list of the **zFAST** commands is available in Section 4.4.8. The following sections provide information for the most common commands.

The content of the script is also presented through various examples.

4.4.1 General syntactical rules for the zFAST script

In the **zFAST** script, the commands require some specific ordering rules but the options for each command can be declared in any order.

The **zFAST** script follows the basic Tcl usage rules:

- Indentation is not meaningful but it makes scripts easier to read.
- Commands are continued on several lines by adding a backslash (\) at the end of a line.
- Comments are preceeded by a sharp symbol (#); it is strongly recommended to keep comments on separate lines (DO NOT mix Tcl commands and comments in the same line of the script).

Instead of commenting out the Tcl commands, it is recommended to do the following:

```
if {0} {
    cmn::option work w1
}
```

Note that the characters around the 0 are curly brackets (and not parentheses).



- It is possible to have interlaced curly brackets in the commands of **zFAST** script, as in the following example:

```
cmn::elaborate {add2 -l lib3 -v rtl} \  
               -force_bb {mod1 mod2 {\a[0] }} \  
               -param add2.MYCONST 0
```

Where the outer set of curly brackets tells Tcl that what is inside is a list, in this case a list of all the modules that are forced to be blackboxes. The inner set of curly brackets is to make sure that the Verilog escaped module name includes the trailing space.

Inner curly brackets also could have been put around the other two module names.

4.4.2 Declaring the Source Files for RTL Analysis

The source files are declared with the command for RTL analysis of the files, which is the first step of the **zFAST** synthesis process. This analyze command is different for Verilog (`vlg::analyze`) and VHDL (`cmn::analyze`) and language-specific options exist.

There can be several analyze commands in the script, with the following recommendations:

- Each analyze command can have several source files, and different options (for different working libraries or different language versions).
- It is recommended to have all the Verilog and SystemVerilog source files declared before the VHDL source files, in particular for VHDL sorting.
- For SystemVerilog source files, it is recommended to have all the source files declared in the same `vlg::analyze` command in order to have all the types of the global namespace visible from all the files.
- When several analyze commands exist for the same language, the performance of the synthesis is lower.

4.4.2.1 RTL Analysis command for a Verilog Source File

A Verilog source file is declared with the command which proceeds with RTL analysis, as in the following example:

```
vlg::analyze <vlog_src_files> [-options]
```

By default, Verilog source files are analyzed as Verilog-2001 (V2K); use `-v95` or `-sv` options to analyze the files with the Verilog-95 or Systemverilog standards.

Examples for the Verilog analysis command are available in Sections 4.4.5, 4.4.6 and 4.4.7.

Table 1: Options for Verilog analysis command

Option	Description
<code>-work <path_to_work_lib></code>	Declaration of the working library.
<code>-v95</code> <code>-v2k (default)</code> <code>-sv</code>	Declaration of the language.



Option	Description
-verilog2001ext .<extension>	Extension of Verilog-2001 files for automatic language detection. Example: -verilog2001ext .v2k
-systemverilogext .<extension>	Extension of the SystemVerilog source files for automatic language detection. Example: -systemverilogext .sv
-v <filename>	Declaration of a Verilog library file to search for definitions of modules and UDP instances.
-y <directory>	Declaration of a Verilog library directory to search for definitions of modules and UDP instances.
-libext	
-incdir <directory>	Declaration of a directory to search for include files.
-libext .<extension>	Declaration of the file name extension to use in the search (with -y option).
-librescan	Forces to start library searches for module definitions from the first library given by a -v option.
-define <macro_name>	Defines a text macro that is used in the RTL source.
-define {<macro_name> <value>}	Sets a value for a macro or replaces a define in the source code.
-attr <attr_name> <value>	Sets a zFAST attribute for the corresponding command, as described in Section 4.4.4.3. Several attributes can be set for the same analysis command.

4.4.2.2 RTL Analysis Command for a VHDL Source File

A VHDL source file is declared with the command which proceeds with RTL analysis, as in the following example:

```
cmn::analyze { [-options] \  
    -vhdl {<vhdl_src_files> [-vhdl_options]} }
```

The options which are applicable for the entire analysis process are added separately from the -vhdl lists ([-options] in the above syntax example), in particular the -sort option for automatic sorting of the VHDL files.

The VHDL files can be declared in several -vhdl lists with their specific options ([-vhdl_options] in the above syntax example), in particular for the working library or the language compatibility.

When the cmn::analyze command has several lists of VHDL source files, all the VHDL source files of all the lists are sorted out when the -sort option is present.

By default, VHDL source files are analyzed as VHDL93 (use -vhdl87 for the corresponding -vhdl list to analyze the file with the VHDL 87 standard).

Examples for the VHDL analysis command are available in Sections 4.4.5 and 4.4.6.



Table 2: Options for VHDL analysis command

Possible [-options]	Description
-vhdl	Mandatory parameter to declare a list fo VHDL source files.
-sort	Automatic sorting of the VHDL source files for the corresponding command.

Possible [-vhdl_options]	Description
-work <path_to_work_lib>	Declaration of the working library.
-vhdl87 -vhdl93 (default)	Declaration of the language (not necessary forVHDL 93).
-attr <attr_name> <value>	Sets a zFAST attribute for the corresponding command, as described in Section 4.4.4.3. Several attributes can be set for the same list of VHDL files.

4.4.3 Elaboration

This command elaborates the whole RTL group, and defines the top name (for Verilog designs) or the configuration (for VHDL designs).

```
cmn::elaborate <top_name>
```

or

```
cmn::elaborate <configuration>
```

The supported options for elaboration in the **zFAST** script are similar to commercial simulators and synthesizers:

- Declaration of blackboxes: -force_bb <module_name>
- Definition of parameters and macros: -param <parameter> <value>

4.4.4 Specific Commands

4.4.4.1 Declaring the default working library

The default working library is declared by adding the following command before any analyze or elaborate command:

```
cmn::option work <path_working_lib>
```

Note that the analyze and/or elaborate commands after this default setting can modify their own working library (with -work option).

4.4.4.2 Adding a **zFAST** script into another one

In a similar manner to the source command in native Tcl, an additional **zFAST** script can be taken into account from the current script by adding the following command:

```
cmn::source <added_script>
```



4.4.4.3 Modifying the **zFAST** attributes in the script

When an **Additional zFAST Attribute File** is declared in **zCui**, the attributes are set for the entire RTL group.

The script mode provides the capability to set the attributes explicitly in the **zFAST** script on a command-by-command basis. When the attributes are set in the script, the syntax is different from the attribute file syntax described in Chapter 7.

In script mode, attributes can be set as options for each command with the following syntax:

```
<command> -attr <attribute_name> <attribute_value>
```

Several attributes can be set for the same command.

The attributes set in the script file override the settings in the **Additional zFAST Attribute File** declared in **zCui**.

Example:

This example sets the obeypragmatranslate attribute for a Verilog analyze command:

```
vlg::analyze -attr obeypragmatranslate true test_src/add2.v
```

4.4.5 **Basic Example of zFAST Script**

This example shows the basic use model of the script mode, which is how to:

- declare the default working library
- launch analysis of the Verilog and VHDL source files
- launch elaboration

```
cmn::option work w1  
vlg::analyze test_src/add2.v  
cmn::analyze {-vhd1 {test_src/adder.vhd}}  
cmn::elaborate add2
```

In the above example:

- w1 is the default working library for all of the analyze commands. As neither of the two analyze commands specify a different working library, w1 will always be used.
- It is a mixed-language design: add2.v is a Verilog file, adder.vhd is a VHDL file. add2 is the top level of the design.
- Elaboration is processed for add2 which is the top of the design.



4.4.6 zFAST Script Example with Options

The following example shows different options for analysis and elaboration of a mixed-language design.

```
cmn::option work w1
vlg::analyze -work lib1 -sv test_src/add1.v test_src/add2.v
vlg::analyze -v95 -work lib3 test_src/add3.v test_src/add4.v
cmn::analyze { -sort \
               -vhd1 {-work lib2 -vhd187\ }
               test_src/adder1.vhd test_src/adder2.vhd}
cmn::elaborate {add2 -l lib3 -v rtl} \
               -force_bb {mod1 mod2 {\a[0] }} \
               -param add2.MYCONST 0
```

In the above example:

- It is a mixed-language design: add1.v and add2.v are SystemVerilog files, adder1.vhd and adder2.vhd are VHDL files and add3.v and add4.v are Verilog 95 files. add2 is the top level of the design.
- w1 is defined as the default working library but actually not used because each analyze command defines its own working library: SystemVerilog files are analyzed in lib1 library; VHDL files are analyzed in lib2 and Verilog95 source files are analyzed in lib3.
- Elaboration is processed in lib3 library.
- mod1, mod2 and \a[0] are declared as blackboxes for synthesis.
- add2.MYCONST parameter is set to 0 with -param option.

4.4.7 zFAST Script Example with Advanced Options for a Verilog design

This example shows Verilog-dedicated options for the analysis command:

```
cmn::option work w1
vlg::analyze
      -verilog2001ext .v -systemverilogext .sv \
      -v lib_file.v -y lib -incdir inc -libext .vlog -librescan \
      -define {MY_DEFINE 1} -define MY_TOGGLE \
      tile.v top.sv
cmn::elaborate add2
```

In the above example:

- The design has 2 Verilog source files: tile.v and top.sv. The top level module is add2.
- w1 is defined as the default working library.
- **zFAST** is configured to analyze automatically the .v files as Verilog2001 files and .sv files as SystemVerilog files.
- Specific options for Verilog libraries and include files are set for analysis.
- MY_DEFINE is set to 1 with -define option.
- MY_TOGGLE is set with -define option.



4.4.8 List of Commands for the zFAST Script

The following table lists all the commands which can exist in the zFAST script. Direct links are added to the general description and to relevant examples in the manual.

Table 3: List of Commands for zFAST Script

Commands	Options	General Description	Example
cmn:source		4.4.4.2	
cmn::option	work <work>	4.4.4.1	4.4.5 4.4.6 4.4.7
vlg::analyze	<file>	4.4.2.1	4.4.5 4.4.6 4.4.7
	-define <macro_name>	4.4.2.1	4.4.7
	-define {<macro_name> <macro_value>}	4.4.2.1	4.4.7
	-incdir <dir>	4.4.2.1	4.4.7
	-libext <ext>	4.4.2.1	4.4.7
	-librescan	4.4.2.1	4.4.7
	-v <lib file>	4.4.2.1	4.4.7
	-y <lib dir>	4.4.2.1	4.4.7
	(-v95 -v2k -sv)	4.4.2.1	4.4.6 4.4.7
	-verilog2001ext .<extension>	4.4.2.1	4.4.7
	-systemverilogext .<extension>	4.4.2.1	4.4.7
	-attr <name> <val>	4.4.2.1 4.4.4.3	
	-work <w>	4.4.2.1	4.4.6
cmn::analyze	-sort	4.4.2.2	4.4.6
	-vhdl {<file> [-vhdl_options]}	4.4.2.2	4.4.5 4.4.6
	-attr <name> <val>	4.4.2.2 4.4.4.3	
	(-vhdl87 -vhdl93)	4.4.2.2	4.4.6
	-work <w>	4.4.2.2	4.4.5 4.4.6
cmn::elaborate	<top_name>	4.4.3	4.4.5 4.4.7
	{ <top_name> [-l <lib>] [-v <view>] }	4.4.3	4.4.6
	-force_bb {<module> [<module> ...]}	4.4.3	4.4.6
	-param <top_name>.<param_name> <value>	4.4.3	4.4.6

5 Running and Monitoring Synthesis

Running and monitoring the **zFAST** synthesis in **zCui** is exactly the same for both standard and script modes.

5.1 Running Synthesis

To run only the synthesis and stop before the back-end compilation, you should select **Design** in the **Target** roll-on menu of the **zCui** toolbar (the default setting when launching **zCui** is **Default Back-End**, whatever it was when the project was previously saved):

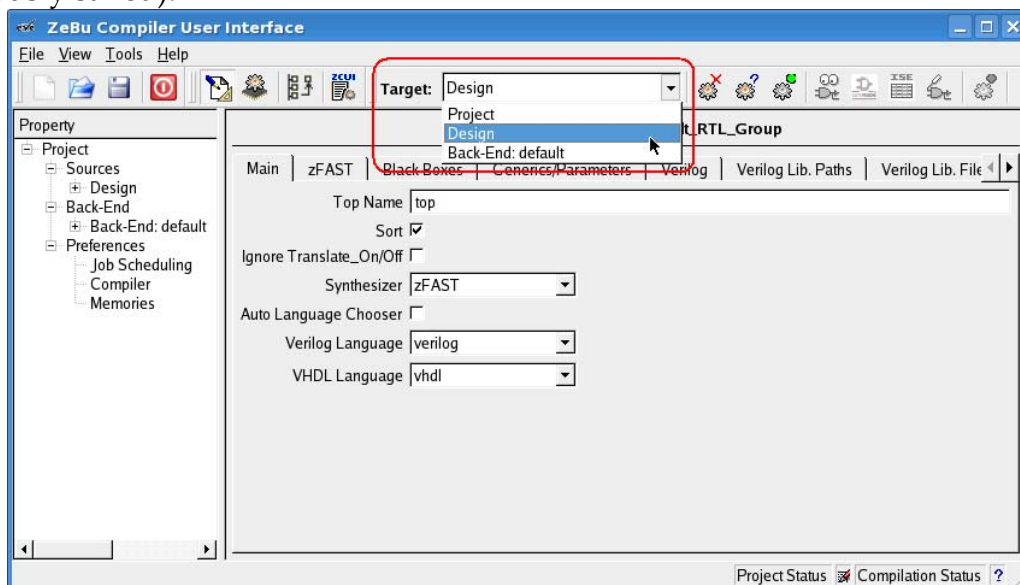


Figure 17: Running zCui for Synthesis only

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

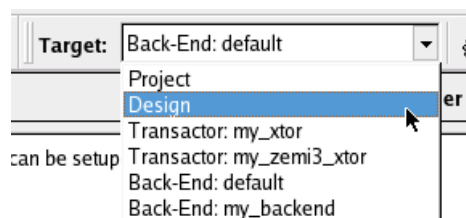


Figure 18: Selecting the compilation target for synthesis

When the selected target is **Design**, the **Clean**, **Evaluate** and **Make** operations are the following ones:

- **Design + Clean:** only the synthesis output files are deleted from the **zCui** output directory. If back-end compilation results exist, they are kept.



- **Design + Evaluate:** the inputs of the project which are necessary to proceed with synthesis are checked.
- **Design + Make:** proceed with synthesis.

5.2 Incrementality

Once the design has been successfully synthesized, launching **zCui** for the Design compilation target only checks if there were any modification in a file declared by the user (source file, RTL-based compilation script, **zFAST** script or additional attribute file) or in the **zCui** settings for synthesis. If no modification is detected, no synthesis task is actually launched.

However, if any of the files declared by the user is modified or if any of the **zCui** settings for synthesis is modified, the elaboration is automatically relaunched. The modules that have changed through elaboration are resynthesized in new bundles; unmodified modules are not resynthesized.

This incrementality feature is also applicable when modifications occur in an include file or in a file that has been picked-up from a library.

5.3 Monitoring the Synthesis

When synthesizing with **zFAST**, the following 3 steps are performed:

- Analysis/Elaboration: the inputs of this step are the user RTL files and user parameters. During this step the RTL sources are analyzed according to the parameters specified in the GUI or in the **zFAST** script (include paths, macros...). This step results in elaborated RTL files. In **zCui**, this analysis/elaboration step is launched as a single process; in the **Task Tree** pane, this step is named **Make RTL FrontEnd**.
- Synthesis: The inputs of this step are the elaborated files generated during analysis/elaboration step. This step performs the RTL to EDIF synthesis. In **zCui**, the synthesis step is launched in one or several processes according to the bundle settings (see Section 2.3.4) and if chunks are used (see Section 8.2); in the **Task Tree** pane, this step is named **Synthesize <top_name>_Bundle_xx**.
- Post-processing: netlist merge for later back-end compilation, processing of **zMem** memories, processing for **zFAST** Stat Browser.

In **zCui**, these steps can be monitored in the **Compilation** view as shown in the following figure:

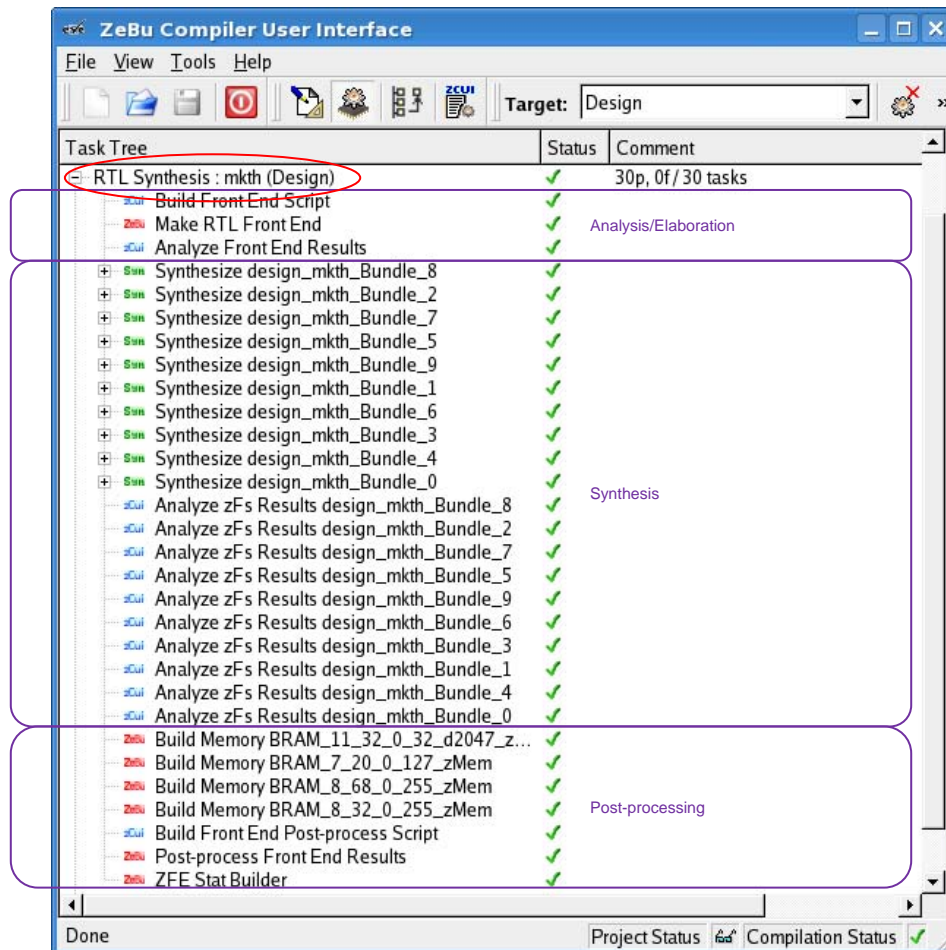


Figure 19: Synthesis steps in Compilation View

Each RTL group has a separate section in the **Task Tree** pane, named **RTL Synthesis: <group name> (<top>)**, which gathers all the steps for analysis/elaboration, synthesis and post-processing for the RTL group.

5.4 zFAST outputs

zFAST outputs are created separately for each RTL group in the following directory: `<zcu.work>/design/synth_<rtlgroupmane>/`, where `<zcu.work>` is the path to the **zCui** working directory (default is `zcu.work`); this path can be changed in the **Project** → **Properties** panel.

This directory includes in particular a memory initialization file named `readmem.dump`. This file can be declared for runtime initialization of the memory, or used as a template to write a memory initialization file, as described in the ZeBu runtime manuals. There is no automatic initialization from this file.

The elaborated source files, the resulting EDIF netlists for back-end compilation and the files for **zMem** memory are stored in separate subdirectories.

5.5 Analyzing the zFAST log files

Log files for each step of the RTL synthesis tasks are displayed in the right pane of the **Compilation** view. Significant information for user is available in particular in the logs of the following steps:

- **Make RTL Front End:** log of the analysis and elaboration.
- **Synthesize <top_name>_Bundle_xx** : one log is available for each bundle.
- **Post-process Front End Results:** log of the post-processing of the synthesized netlists.

zCui provides search capability in each log which is interesting to investigate analysis/elaboration and post-processing steps, as described in Section 5.5.1.

For the synthesis logs, **zCui** incremental process and the multiple bundles made it impossible to search efficiently through the **zCui** interface. When investigating synthesis issues, it is necessary to search manually from the shell (with the `grep` command for example) in the directory which includes all the `<module_name>.log` files:

```
<zcu.work>/design/synth_<rtlgroupname>/synth_log_dir/
```

Where `<zcu.work>` is the path to the **zCui** working directory which is `zcu.work` by default; this path can be changed in the **Project** → **Properties** panel when creating the compilation project.

5.5.1 Search Capability in zCui

The **zCui** log viewer provides search capability with different criteria. The toolbar at the bottom of the 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 log file.

To change the active search toolbar, use the up-down arrows placed next to the search criteria icon (here the “Search for Text” criterion is selected):





5.5.2 Investigating Memories inferred with zMem

When a design memory is inferred with **zMem**, the log file of the **Synthesize <name>** step includes the following information:

```
# step INFO : Found a var. addr. write of memory <memory_elaborated_name>, will not bit-  
blast  
# step INFO : Got memory <memory_elaborated_name> with <n_read> read port(s), <n_write> write  
port(s), and <n_readwrite> RW port(s)
```

Searching for “Got Memory” is helpful.

5.5.3 Analyzing the Post Processing Log File

The log file of the **Post Processing Front-End Results** step provides in particular the following information:

- List of blackboxes in the output netlist of the synthesizer (search for “step show_black_box” in the log):

```
# step show_black_box : black box found : 'global_main.i_manual_bram' (module : 'manual_bram')
```

- Post-synthesis gate count (search for “step STATS” in the log): size of the output netlist of the synthesizer; cannot be used to report an optimized version of the gate count (only available after the **Build System** step during the back-end compilation).

```
# step STATS : | reg| lut| bram|memsize|m18x18|bufg |buft |nbI/O |nbBB|nbME|dsp48| name  
# step STATS : -----  
# step STATS : | 224|20959| 0| 0| 4 | 0 | 0 | 2487 |552 | 0 | 0 | global_main
```

5.6 Proceeding with Back-End Compilation

Once the design has been successfully synthesized with **zFAST**, you can proceed with the back-end compilation which is fully integrated in **zCui** as well. The [ZeBu Compilation Manual](#) provides appropriate information about the complete compilation flow for ZeBu.

6 Language-specific Features

6.1 Verilog and SystemVerilog Specific Features

6.1.1 Definition Rules for Blackboxes

When a module is empty in the RTL source files, **zFAST** automatically synthesizes it as a blackbox without any additional declaration.

However, any other module of the RTL source files can be transformed into a blackbox by the synthesizer if it is declared correctly for that purpose:

- When synthesizing in standard mode, add the module name in the **Black Boxes** tab of **RTL Group Properties**.
- When synthesizing in script mode, the `-force_bb` option has to be added for the elaboration command, as described in Section 4.4.3.

If preferred (but not for the same modules), this declaration can be done via the following attribute in the **Additional zFAST Attribute File**:

```
Compile:ForceBlackBox=(<list of module or entity names>)
```

Where the items of the list are separated with a space character (parentheses are mandatory when several modules/entities are listed).

Example:

```
Compile:ForceBlackBox=(mod1 mod2)
```

6.1.2 pullup/pulldown Verilog Primitives

The pullup and pulldown Verilog primitives are supported when synthesizing with **zFAST**.

The tristate buses connected to these primitives will be processed correctly without any additional declaration for back-end compilation.

6.1.3 tri0/tri1/trireg wires

tri0 and tri1 wires are supported when the `Compile:ConvertTri` attribute is set to true in the **Additional zFAST Attribute File**. In such a case, the trireg wires are synthesized as standard wires. The wires will be processed correctly without any additional declaration for back-end compilation.

If this attribute is not set to true, such wire cause a fatal error during synthesis.

6.1.4 Verilog strength

Verilog strength values are ignored when synthesizing with **zFAST**.

6.1.5 pmos/nmos primitives

The pmos and nmos primitives are supported by **zFAST** when the `Compile:ConvertMOS` attribute is set to true in the **Additional zFAST Attribute**



File. In such a case, pmos/nmos primitives are converted to bufif0/bufif1 and the strength value is ignored.

If this attribute is not set to true, such wire cause a fatal error during synthesis.

6.1.6 #delay statements

The #delay statements are ignored when synthesizing with **zFAST**.

6.1.7 \$display & others Verilog tasks

\$display and other Verilog tasks are ignored when synthesizing with **zFAST**.

6.1.8 Memory Initialization with \$readmemb/\$readmemh

\$readmemb and \$readmemh are supported inside initial blocks and a file (readmem.dump) is automatically generated by **zFAST** with memory initialization data for runtime purposes.

The file is stored in the <zcu.work>/design/synth_<rtlgroupname>/ directory and it can be declared as a memory initialization file for runtime.

6.1.9 Memory inference

6.1.9.1 Memories in Verilog & SystemVerilog

In Verilog & SystemVerilog, a memory is declared as an array:

```
reg [7:0] mem [127:0];
```

In such a case, **zFAST** is able to infer a **zMem**-based memory, which will support read/write access at runtime.

In SystemVerilog, memories can be described as multi-dimensional arrays:

```
reg [7:0][2:0] mem [127:0][3:0];
```

Such memories also support read/write accesses at runtime. The data representation in the ZeBu runtime software matches the file format described in the SystemVerilog Language Reference Manual.

6.1.9.2 Size threshold for Memory Inference

A memory can be inferred as logic gates or as **zMem** depending on a size threshold which is declared in the **zFAST** tab of **zCui** and is applicable globally for the RTL group. This threshold is defined in bits and corresponds to the total size of the memory:

- Any memory smaller than the threshold is implemented as logic gates.
- Any memory bigger than the threshold is implemented with a **zMem**-based memory model.
- Some specific constraints on the number or type of ports of a memory may force the implementation with **zMem** although the memory is smaller than the threshold.

The default threshold is 2,048 bits.



It is possible to force the inference of a design memory as logic or **zMem**-based by adding the following attribute in the **Additional zFAST Attribute File**:

- To force a memory to be implemented as logic:
`compile:implementAsFlops=(<memory_paths_list>)`
- To force a memory to be implemented with **zMem**:
`compile:implementAsMemory=(<memory_paths_list>)`

Where `<memory_paths_list>` is a list of paths to memory instances; each path in the list can be of type `<full_path_from_top>.<mem_instance>` or `<module>.<mem_instance>`.

6.1.9.3 **zMem** memory types

When a **zMem** memory is inferred, **zMem** is in auto mode by default, which means that the decision for the memory implementation (LUT-RAM based, BRAM-based or zrm-based memories) is taken automatically. Some settings in the **Preferences** → **Memories** panel of **zCui** may impact the automatic implementation.

The type of the **zMem** memory can be forced in **zFAST** to LUTs, BRAMs or zrm via an attribute in the additional command file:

```
MemoryType:lut = ( <memory_paths_list> )
MemoryType:ramb = ( <memory_paths_list> )
MemoryType:zrm = ( <memory_paths_list> )
```

Note that `<memory_paths_list>` can be a full path from the top to the array name or under the form `<module>.<array_name>`.

6.1.9.4 Optimizing memory synthesis

The optimal synthesis of a memory highly depends on the coding style of the memory. In some cases the result is sub-optimal.

The result of memory inference can be analyzed with the **zFAST** Stat Browser described in Section 8.3. The following information should be checked:

- The number of ports of the memory.
- The synchronism/asynchronism of the ports.
- The bit-enable/byte-enable information.

If a memory has more than 128 ports, **zFAST** stops with an error because it is very uncommon to have such a high number of ports and because a memory with too many ports requires a bigger logic size, impacting the achievable runtime performance.

However, the number of ports which triggers this error can be changed with the following attribute:

```
Compile:PortLimit=<new number>
```

If you get a memory with more ports than originally expected, you can either modify the RTL source code or change the following **zFAST** attributes:

Table 4: zFAST Attributes to optimize design memories

Attribute	Description
Compile:SplitHorizontally	<p>Splits memories with constant subsets of address bits. Default is false.</p> <p>When set to true, the logic size is optimized by splitting the memory on multiple zMem-based memories. Note that the resulting memory cannot be accessed at runtime for read or write operations.</p> <p>Example:</p> <pre>reg [7:0] mem [127:0][1:0][127:0]; always @(posedge clk) begin mem[addr1][1][addr2] = din; dout = [addr1][1][addr2]; end</pre>
Compile:DropWriteOnlyMemories	When set to true, write-only memories are not synthesized. Default is false.
MemoryType:OptimizeBie=(<memory_paths_list>)	Activates the optimized bit-enable mode of zMem for the listed memories.

Other attributes to control memory synthesis:

Attribute	Description
Compile:MakeSyncWrites = { false posedge negedge dual }	<p>Set to posedge, this attribute transforms asynchronous writes into synchronous writes using the posedge of write enable as port clock:</p> <pre>always_comb if (en) mem[addr] = ...</pre> <p>treated as</p> <pre>always @(posedge en) mem[addr] = ...</pre> <p>Set to negedge this attribute transforms asynchronous writes into synchronous writes using the negedge of write enable as port clock:</p> <pre>always @(negedge en) mem[addr] = ...</pre> <p>Set to dual this attribute transforms asynchronous writes into synchronous writes using negedge and posedge of write enable as port clock:</p> <pre>always @(en) mem[addr] = ...</pre>
Compile:InlineReadmem=(<memory_paths_list>)	<p>Puts directly the content of a readmem file into Read Only Memories synthesized as logic for the listed memories.</p> <p>If a Verilog task \$readmemb or \$readmemh is called in an initial</p>



Attribute	Description
	block and the memory init file is accessible at synthesis time, then the content of the file is inlined in the netlist. The memories processed this way will be unloadable at runtime (The content of the memory cannot be changed at runtime).

6.1.10 Non-RTL language support

6.1.10.1 Hierarchical references

The following Verilog hierarchical references (hrefs) can be synthesized with **zFAST**:

- Absolute hierarchical references (from the top)
- Relative hierarchical references (from the current scope)
- Hierarchical references to VHDL when the `Compile:HrefToVHDL` attribute is set to `true` in the **Additional zFAST Attribute file** in **zCui**.
- Hierarchical references to memories, assuming that such hrefs add ports to the memories.

6.1.10.2 SystemVerilog bind statement

SystemVerilog `bind` statement can be declared in the global scope or inside a module.

6.1.10.3 DPI Calls in the source files

As described in the dedicated Application Note (AN029), the zDPI feature supports “streaming” DPI import calls in the DUT.

6.1.10.4 SystemVerilog Assertions

A large subset of SVA language can be synthesized with **zFAST**, as described in the dedicated Application Note (AN028).

6.2 Synthesis Pragmas

The following pragmas are always processed:

VHDL

```
--eve translate_off  
--eve translate_on
```

Verilog

```
/* eve translate_off */  
/* eve translate_on */  
  
// eve translate_off  
// eve translate_on
```

The following pragmas are processed only if the `compile:obeysynopsystranslate` attribute is set to `true`:

VHDL

```
--synopsys translate_off  
--synopsys translate_on
```

Verilog

```
/* synopsys translate_off */  
/* synopsys translate_on */  
  
// synopsys translate_off  
// synopsys translate_on
```




The following pragma are processed only if the `compile:obeypragmatranslate` attribute is set to true:

VHDL

```
--pragma translate_off/on  
--pragma translate_on
```

Verilog

```
/* pragma translate_off */  
/* pragma translate_on */
```

```
// pragma translate_off  
// pragma translate_on
```



7 Writing an Attribute File for zFAST

This chapter describes the syntax to modify the **zFAST** settings through a dedicated “Attribute File”. In both standard mode and script mode, this file is declared in the **zFAST** tab, in the **Additional zFAST Attribute File**. Such a file is applicable to **zFAST** for the entire RTL group.

In script mode, some attributes can also be set explicitly in the **zFAST** script for a given command, as described in Section 4.4.4.3. Note that the syntax for the **zFAST** script is not the same as for the Attribute File.

7.1 Syntax Rules

Table 5: Syntax Rules for zFAST Attribute File

Syntax Rules	Example
The syntax is always in the form of a pair of name/ value, as in the following example	Hcs:SVA = true
Values are strings or lists of strings (lists have parenthesis, and list items are separated with spaces)	DefaultLength = 20 DefaultPath = (here there "every where?")
Names are case insensitive	DefaultLength = 20 defaultLENGTH = 20 defaultlength = 20
Last/Latest assignment wins	X = 10 Y = 20 X = 30 ← The application will use this value of X
Values can be locked by adding @	@X = 10 X = 20 ← This raises an error
Values can be substituted	Arch = x86_64 PATH = "\${ZEBU_ROOT}/bin/\${Arch}" PATH = "\${ZEBU_ROOT}/bin/\${Arch?x686}"
Values can be appended	X = 10 +X = 20 ← X == (10 20)
Values can be concatenated:	X = "Hello" &X = " World" ← X == "Hello World"
Several boolean values are supported for true	true, yes, on, y, t, 1



7.2 List of attributes

Table 6: List of zFAST Attributes

Type	Attribute Name	Default Values	Description	See Section
MEMORY SYNTH CTRL	Compile:MakeSyncWrites = {false true posedge negedge dual}	false	Asynchronous writes are made synchronous to an edge (or both) of write-enable signal	6.1.9.4
MEMORY SYNTH CTRL	Compile:PortLimit=<number>	128	If the number of ports is larger than <number> then a fatal error is thrown	6.1.9.4
MEMORY SYNTH CTRL	Compile:InlineReadmem=(<memory_paths_list>)	N/A	Inlines content of the readmem file into the flop memory	6.1.9.4
MEMORY SYNTH CTRL	Compile:DropWriteOnlyMemories	false	Removes write only memories	6.1.9.4
MEMORY SYNTH CTRL	MemoryType:RAMLUT = (<memory_paths_list>)	N/A	Forces implementation of the memories in the list as LUTs	6.1.9.3
MEMORY SYNTH CTRL	MemoryType:BRAM = (<memory_paths_list>)	N/A	Forces implementation of the memories in the list as BRAMs	6.1.9.3
MEMORY SYNTH CTRL	MemoryType:zrm = (<memory_paths_list>)	N/A	Forces implementation of the memories in the list as zrm memories	6.1.9.3
MEMORY SYNTH CTRL	Compile:ImplementAsFlops=(<memory_paths_list>)	N/A	Forces implementation of the memories in the list as logic	6.1.9.2
MEMORY SYNTH CTRL	Compile:ImplementAsMemory=(<memory_paths_list>)	N/A	Forces implementation of the memories in the list as zMem memories	6.1.9.2
MEMORY SYNTH CTRL	MemoryStyle:ReadBeforeWrite =(<memory_paths_list>)	N/A	Forces ports of zMem memories to be read-before-write	9
MEMORY SYNTH CTRL	MemoryStyle:ReadAfterWrite= (<memory_paths_list>)	N/A	Forces ports of zMem memories to be read-after-write	-
MEMORY SYNTH CTRL	Compile:SplitHorizontally	false		6.1.9.4
MEMORY SYNTH CTRL	MemoryType:OptimizeBie=(<memory_paths_list>)	N/A	Activates the optimized bit-enable mode of zMem for the listed memories.	6.1.9



Type	Attribute Name	Default Values	Description	See Section
MEMORY SYNTH CTRL	Compile:SliceThreshold=<number>	20	If a memory with more ports than <number> has a constant address range and the splitHorizontally attribute is set to true, then the memory is split vertically. Needs to be handled by runtime.	-
PARSE CTRL	Compile:ObeySynopsysTranslate	false	Obeys synopsys translate on/off pragma	6.2
PARSE CTRL	Compile:ObeyPragmaTranslate	false	Obeys pragma translate on/off pragma	6.2
LOGIC INFERENCE CTRL	Compile:Expand:DSPMultthreshold=<threshold>	16	If the number of bits of the two operands of a multiplier is larger than <threshold>, DSPs are inferred instead of LUTs	-
LOGIC INFERENCE CTRL	Compile:MaxLoopIterations=<max>	2048	Modifies the maximum number for loops in the design (an error is thrown if a loop has more iterations)	-
LOGIC INFERENCE CTRL	Compile:ForceBlackBox=(<list of module/entity names>)	N/A	Forces the target modules to become blackboxes in output of synthesis	6.1.1
LOGIC INFERENCE CTRL	Compile:ConvertTri	false	When set to true, supports synthesis of tri0/tri1 wires as tristate with pull-down or pull-up resolution; trireg wires are as standard wires.	6.1.3
LOGIC INFERENCE CTRL	Compile:ConvertMOS	false	When set to true, supports synthesis of pmos/nmos primitives as buif0/buif1.	6.1.5
MISC	Compile:HrefToVHDL	false	When set to true, zFAST supports hierarchical references to VHDL in the Verilog sources.	6.1.10.1

8 Advanced Usage

8.1 RTL-based Compilation Scripts

8.1.1 Introduction

The entire ZeBu compilation flow supports the declaration of hierarchical paths to nets and instances with the name used in the RTL source files. This is applicable for the following features:

- Declaration of probes (probe_signals command), whatever their types: dynamic, flexible, value-change or c-calls.
- Declaration of tristate signals for back-end processing (tristate command).
- Declaration of design signals to be forced by the compiler (force command).
- Declaration of signals for dynamic force (force_dyn command).

All these declaration are done in one or several Tcl files, declared in the **Design → RTL-based Compilation Scripts** item of the **Project Tree** pane of **zCui**.

The declaration based on RTL paths is applicable for some elements in the following system-level compiler commands:

Command/Option	EDIF	RTL
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>
tristate	<signal>	-rtlname <rtl_signal>
force	-pin <port> -net <net> -pin_input <port> -pin_output <port> -source_pin <port> -source_net <net>	-rtlname <rtl_port> -rtlname <rtl_net> -rtlname_input <rtl_port> -rtlname_output <rtl_port> -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>

Note that options not listed in this table are not impacted by or do not support the declaration of RTL paths (in particular -module option).

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 RTL paths for the interface of top.instance:

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



Declaring flexible probes with RTL paths for a `foo` signal in `top.instance`:

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

Declaring a flexible probe with a pattern-matching RTL path:

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

Declaring a flexible probe 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 and EDIF paths with the former syntax but mixing both RTL and EDIF paths may be difficult for legibility.
- Existing EDIF-based declarations for the back-end compiler are still supported but should not be used simultaneously in RTL-based scripts.
- The *ZeBu-Server Compilation Manual* has not been updated for this feature. The syntax of these commands in Revision b is still applicable, but the files in which these commands are declared are no longer correct for use with RTL paths (table in Section 3.7 of the manual is only applicable for EDIF paths).

8.1.2 Limitation for Escaped RTL Identifiers

The ZeBu RTL-based compilation scripts support standard Verilog identifiers and escaped identifiers:

- A standard Verilog identifier is any sequence of letters, digits, dollar signs (\$), and underscore (_) symbol, except that the first must be a letter or the underscore; the first character may not be a digit or \$. Upper and lower case letters are considered as different characters.
- Escaped identifiers start with the backslash character (\) and may include any printable ASCII character. An escaped identifier ends with white space. The leading backslash character is not considered to be part of the identifier.

When using an escaped Verilog identifier, the curly brackets are mandatory to match the Tcl constraints because of the backslash character with a space. If the curly brackets are not added for escaped identifiers, the space is interpreted as a list separator by Tcl.

The following limitations apply for all the commands supported in the RTL-based compilation script:

- Curly brackets can not be used in the RTL identifier because they would cause conflicts between the backslashed characters. A possible workaround is to use the EDIF name instead of the RTL identifier for nets or instances with curly brackets in their RTL name.
- The characters used for pattern matching, such as * or ?, cannot be used in a Verilog identifier when `-fnmatch` option is used.



Example:

In the following example, the `probe_signals` command uses the `-rtlname` option for lists (the corresponding Verilog source code is available below):

```
probe_signals -type dynamic -rtlname {{top.modgen.\loopi[0]
[0].loopj[0].a} {top.modgen.\loopi[0] [0].loopj[0].s.i}}
probe_signals -type dynamic -rtlname {{top.modgen.\loopi[0]
[0].loopj[0].\s[0] .i}}
probe_signals -type dynamic -rtlname {{top.modgen.\loopi[0]
[0].loopj[1].st[1].a}}
probe_signals -type dynamic -rtlname {{top.modgen.\loopi[0]
[1].loopj[0].\st[0] [1].a}}
```

```
module sub(input i, output o);
    assign o = i;
endmodule
```

```
module top(input clk, input [31:0] in, input [31:0] out);

    typedef struct {
        bit a;
    } mystr;

    generate begin : modgen // named generate
        genvar i,j;
        for(i=0;i<2;i++) begin : \loopi[0]
            for(j=0;j<2;j++) begin : loopj
                reg a;
                mystr st [1:0];
                mystr \st[0] [1:0];

                sub s (.i(in[i*2+j]), .o(out[i*2+j]) );
                sub \s[0] (.i(in[i*2+j+5]), .o(out[i*2+j+5]));
            end
        end
    end
endgenerate

endmodule
```




8.1.3 Support of generate blocks

RTL instances declared in generate blocks in the design do not have explicit RTL paths before the source files are elaborated. For such instances, **zFAST** provides the following naming rules:

- When the generate block is named, the instances declared in it use its name, as in the following example:

```
generate begin : mygen
  genvar i;
  for(i=0;i<3;i++) begin: second_level
    reg a;
  end
endgenerate
```

The hierarchical path to a is:

```
top.mygen.second_level [0].a
top.mygen.second_level [1].a
top.mygen.second_level [2].a
```

- When the generate block is not named, the instances declared in it use genblk as a template name, as in the following example

```
generate begin : mygen
  genvar i;
  for(i=0;i<3;i++) begin
    reg a;
  end
endgenerate
```

The hierarchical path to a is:

```
top.mygen.genblk [0].a
top.mygen.genblk [1].a
top.mygen.genblk [2].a
```

8.2 Chunking

In top-down synthesis, the synthesis is by default done in one process for the whole design. In this case **zFAST** flattens the output netlist.

In order to be able to synthesize the design in parallel synthesis using bundles, the design can be splitted into chunks which are several parts of the design hierarchy that can include several levels. Each chunk results into a flatten module in the output netlist of the synthesis.

8.3 Using zFAST Stat Browser

Once an RTL group has been synthesized with **zFAST**, either in standard or script mode, the **zCui** interface offers the capability to analyze the results of the **zFAST** synthesis in a graphical interface known as the **zFAST** Stat Browser.

Note that this section provides information about basic usage of the tool. Additional information will be provided in a future revision of the document.

8.3.1 Launching the zFAST Stat Browser from zCui

This tool is available in the contextual menu of the **RTL Group Properties** item in the **Project Tree** pane, as shown on the following figure:

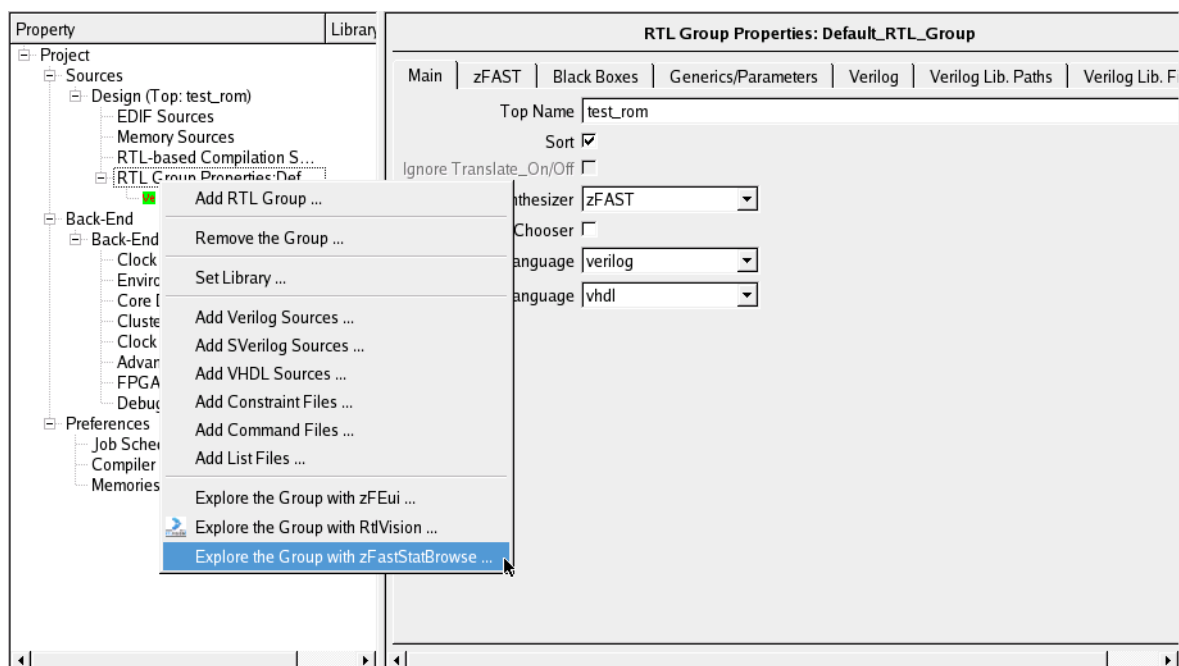


Figure 20: Accessing the zFAST Stat Browser from zCui

When selecting **Explore with zFASTStatBrowse...**, the **zFAST** Stat Browser comes up in a separate window and shows the following opening window:

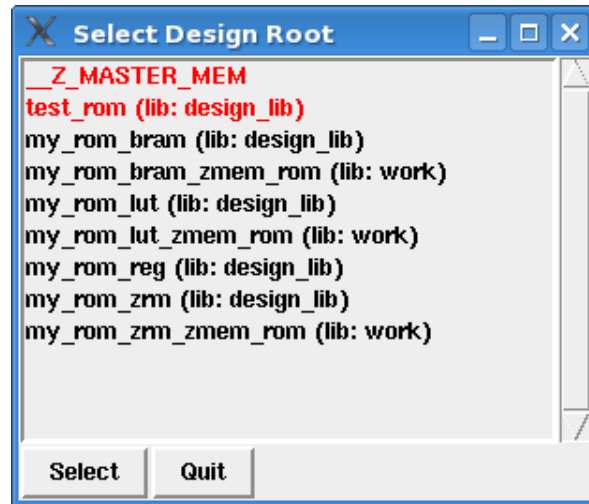


Figure 21: zFAST Stat Browser Opening Window

The top level of the design (`test_rom` in Figure 21) is listed in red; the `__Z_MASTER_MEM` item also displayed in red is a shortcut which gathers all and only the memories and may be of interest when investigating memory inference for a large design.

The items listed in black are the modules and memories instantiated in the design.

8.3.2 Available Statistics for the design

Different statistics are available for the modules and **zMem** memories instantiated in the design.

For modules, the following information about the module itself is available:

- **# Inst** total number of instances of the module in the design hierarchy
- **Self Weight** weight of the module itself, without the weight of instances in it
- **Full Weight** weight of the module including the instances in it.
- **DC Weight** = nb of instances of the module * self weight
- **% Weight** = full weight of the module / weight of the design

Some information about the content of the module is also available:

- **state** number of registers and latches
- **lut.2/3/4/5/6** number of LUT2/LUT3/LUT4 LUT5 and LUT6
- **muxcy** number of muxcy
- **dsp** number of DSPs (not visible on Figure 25)

8.3.3 Browsing the design to get statistics

In order to browse the synthesis results for the entire design, double click on the top level name in the opening window (Figure 21) in order to have the main pane of the **zFAST** Stat Browser:

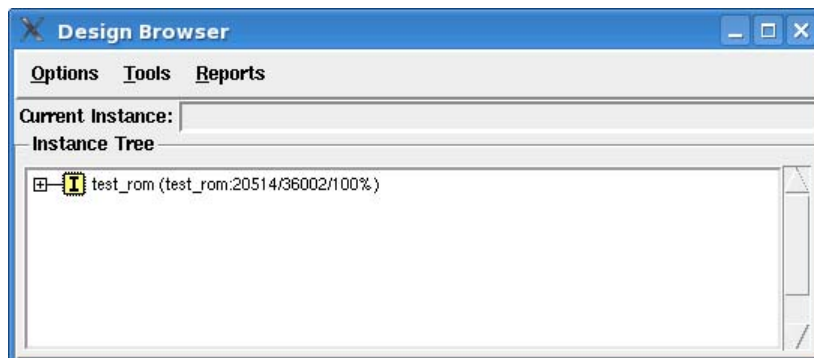


Figure 22: zFAST Stat Browser Main Window

In this **Main** window, you can expand the hierarchy of the design to select any module instance (marked with **I**) or **zMem** memory instances inferred by **zFAST** (marked with **M**) of the design either by clicking on the expand/collapse icon (+/-) or in the contextual menu of the top-level or the module (**Open/Close** items):

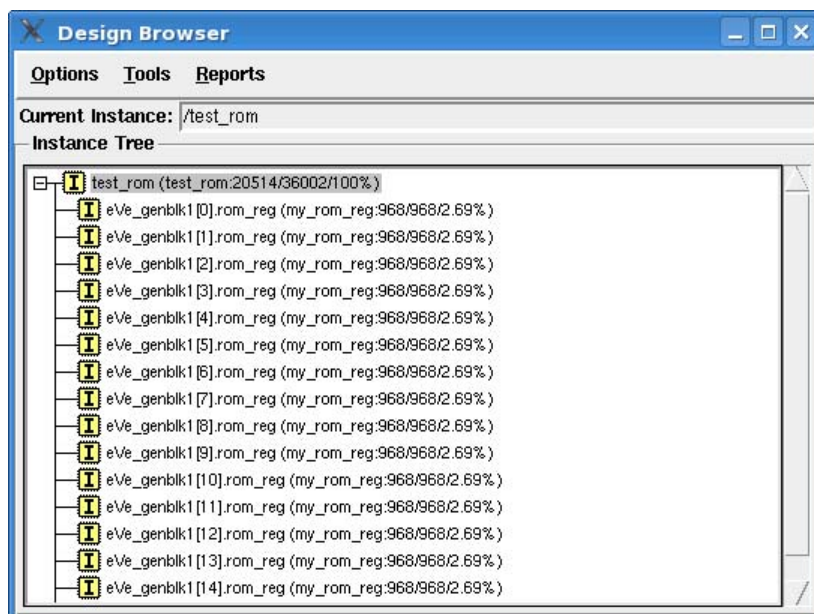


Figure 23: zFAST Stat Browser Main Window with expanded hierarchy

For each instance in the design (marked with **I**), the following information is displayed (see description of the module spreadsheet on the next page for details):
<instance_name> (<module_name>:<module self weight>/<module total weight>/<ratio of the module total weight in the design>)

For each memory in the design (marked with **M**), the following information is displayed:

<memory_name> (<module_name>) [<#read ports> : <#write ports> : <#read-wrtie ports>]

Example:

In Figure 23 for example, `eVe_genblk1[0].rom_reg` is an instantiation of module `my_rom_reg` which weight is 968 in the module itself (self weight) and the weight with all sub-levels is 968 (total weight). In comparison to the weight of the design, the weight ratio for this instance is 2.69%.

The **Details** item in the contextual menu available for the top-level and for each instance of the design provides the following information (see Section 8.3.2 for a description of the items listed this window):

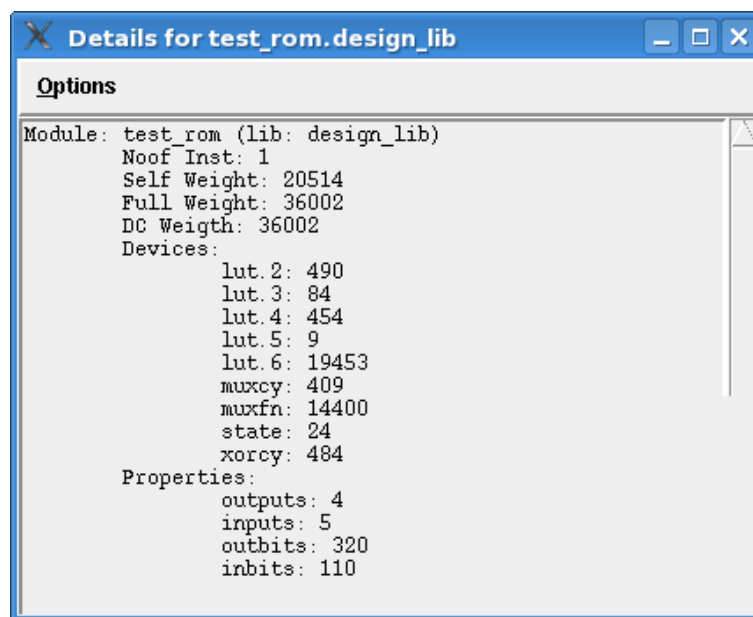


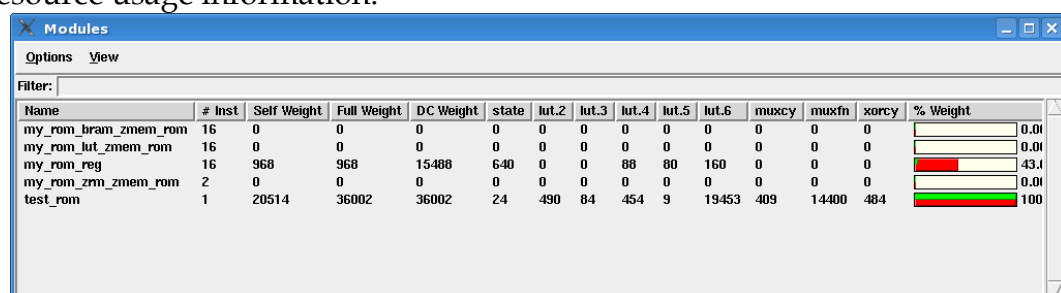
Figure 24: zFAST Stat Browser Details Window

In the **Options** menu, only the **Quit** item is of interest. Other items are reserved for EVE internal usage and are not described here.

8.3.4 Statistics Spreadsheets

In the menu bar of the **Main** window, the **Tools** menu provides items to access resource usage spreadsheets for the module or memory selected in the list:

- When selecting **Modules**, a spreadsheet displays the following weight and resource usage information:

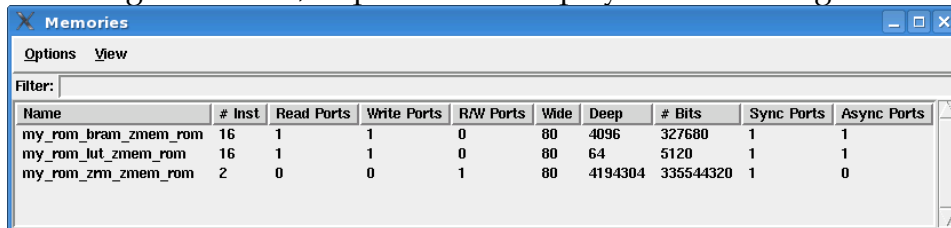


Name	# Inst	Self Weight	Full Weight	DC Weight	state	lut.2	lut.3	lut.4	lut.5	lut.6	muxcy	muxfn	xorcy	% Weight
my_rom_bram_zmem_rom	16	0	0	0	0	0	0	0	0	0	0	0	0	0.01
my_rom_lut_zmem_rom	16	0	0	0	0	0	0	0	0	0	0	0	0	0.01
my_rom_reg	16	968	968	15488	640	0	0	88	80	160	0	0	0	43.1
my_rom_zm_zmem_rom	2	0	0	0	0	0	0	0	0	0	0	0	0	0.01
test_rom	1	20514	36002	36002	24	490	84	454	9	19453	409	14400	484	100

Figure 25: zFAST Stat Browser - Statistics Spreadsheet for Modules

This spreadsheet only shows the columns for resources which are actually used. See section 8.3.2 for a description of each label.

- When selecting **Memories**, a spreadsheet displays the following information:



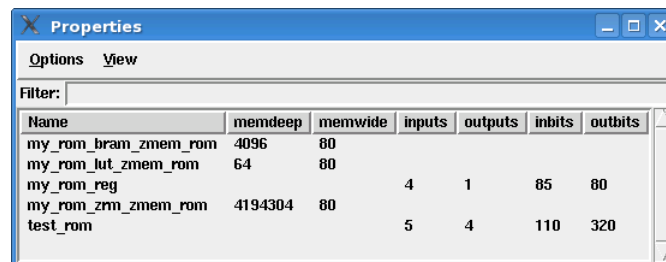
Name	# Inst	Read Ports	Write Ports	R/W Ports	Wide	Deep	# Bits	Sync Ports	Async Ports
my_rom_bram_zmem_rom	16	1	1	0	80	4096	327680	1	1
my_rom_lut_zmem_rom	16	1	1	0	80	64	5120	1	1
my_rom_zm_zmem_rom	2	0	0	1	80	4194304	335544320	1	0

Figure 26: zFAST Stat Browser - Statistics Spreadsheet for Memories

This spreadsheet is very useful to investigate which memories have a very high number of ports because they may be due to inappropriate inference by the synthesizer.

Note: a threshold is set to stop synthesis if a memory has too many ports (by default, any memory with more than 128 ports will cause a synthesis error). This threshold can be modified from **zCui** or with a **zFAST** attribute, as described in Section 7.2.

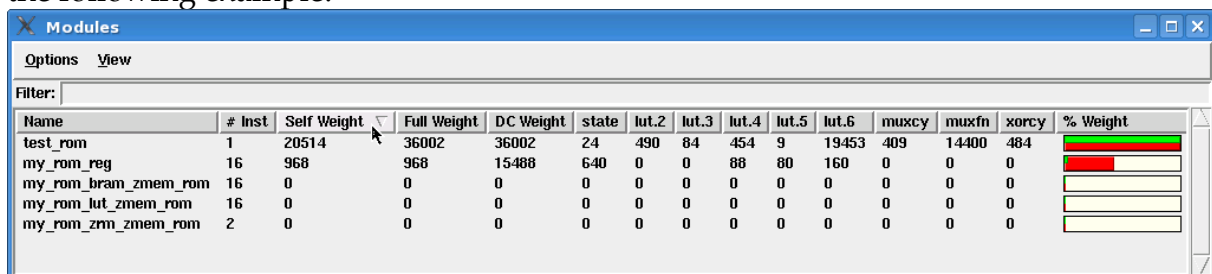
- Properties:** IO properties (pins and bits) for modules as well as memory information for memories:



Name	memdeep	memwide	inputs	outputs	inbits	outbits
my_rom_bram_zmem_rom	4096	80				
my_rom_lut_zmem_rom	64	80				
my_rom_reg			4	1	85	80
my_rom_zm_zmem_rom	4194304	80				
test_rom			5	4	110	320

Figure 27: zFAST Stat Browser - Tools → Properties

The default display is in alphabetical order, as shown in Figure 25, but the spreadsheets can be sorted on any of the columns by clicking on header labels, as in the following example:



Name	# Inst	Self Weight	Full Weight	DC Weight	state	lut.2	lut.3	lut.4	lut.5	lut.6	muxcy	muxfn	xorcy	% Weight
test_rom	1	20514	36002	36002	24	490	84	454	9	19453	409	14400	484	<div style="width: 100%; height: 10px; background-color: red;"></div>
my_rom_reg	16	968	968	15488	640	0	0	88	80	160	0	0	0	<div style="width: 100%; height: 10px; background-color: red;"></div>
my_rom_bram_zmem_rom	16	0	0	0	0	0	0	0	0	0	0	0	0	<div style="width: 100%; height: 10px; background-color: yellow;"></div>
my_rom_lut_zmem_rom	16	0	0	0	0	0	0	0	0	0	0	0	0	<div style="width: 100%; height: 10px; background-color: yellow;"></div>
my_rom_zm_zmem_rom	2	0	0	0	0	0	0	0	0	0	0	0	0	<div style="width: 100%; height: 10px; background-color: yellow;"></div>

Figure 28: zFAST Stat Browser - Sorted-out Statistics Spreadsheet



8.3.5 Additional Features

8.3.5.1 Export Features

The **zFAST** Stat Browser also provides several export features so that the data can be used externally. For any spreadsheet window, the **Export** menu provides the following items:

- **Tab Sep. List:** Dumps the spreadsheet to file as a tab separated list.
- **Colon Sep. List:** Dumps the spreadsheet to file as a ':' separated list.
- **Choose Sep. List:** Dumps the spreadsheet to file as a separated list where the user specifies the separator.
- **Formatted Text:** Dumps the spreadsheet to file as formatted text (recommended to print out the data).
- **To Report Window:** Dumps the spreadsheet as text to a separate window from where it can be cut & pasted etc.

Note that **IAF Attributes** item should not be used with the present software release.

8.3.5.2 Report Selection

In the **Reports** menu of the **Main** window, several reports are available:

- **Top 10 Modules:** report on the top 10 modules in several categories such as Full Weight, Self Weight, State Devices, etc.
- **Top 10 Memories:** Report on the top 10 memories in several categories.
- **Device Summary (All):** Device use summary per device type (LUT2, muxcy etc)
- **Device Summary (By Class):** Device use summary per device class (LUT, MUX, DSP, etc)
- **Module Info:** Per module details.
- **Memory Info:** Per memory details.

By default, these reports are displayed in a dedicated window. However, it is possible to force the output directly in a file by selecting **Set Outfile**. In such a case, all the reported data are stored in the output file and the report window is not opened. The **Close Outfile** item puts the report back in a dedicated window.

It is possible to have all the 6 reports displayed/stored at once by selecting the **All** item, or to select which of the 6 reports are displayed/stored by selecting the **Select** item.

9 Limitations

- Top-Down Synthesis:
 - The generated netlist is flattened or partially flattened. That can introduce clustering issues since clustering is based on hierarchy.
 - The runtime accessibility is reduced: only registers can be accessed in the runtime database.
 - Hierarchical references are not supported

- Memories described in SystemVerilog source files can only have the Read-After-Write priority when processed by **zMem**.

As a workaround, it is possible to force the R/W priority of ALL the ports of the memory in an **Additional zFAST Attribute File**. The following syntax should be used:

```
MemoryStyle:ReadBeforeWrite = (<memory_path_list>)
```

Where <memory_path_list> lists the memory instances which will have Read-Before-Write priority.

Example:

```
MemoryStyle:ReadBeforeWrite = (top.mem mod.mem1)
```

- When several RTL groups are declared for the project, the following limitations apply:
 - Hierarchical references, SystemVerilog Assertions, zDPI feature and RTL-based compilation scripts are not supported.
 - The declaration of a top name is mandatory for each group.
- The top of the design cannot have a Verilog escaped identifier.

10 ZeBu-Server Documents

For each version, the *ZeBu-Server 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-Server documentation package (some are generic manuals for the ZeBu range):

- [1] The *ZeBu-Server Installation Manual* describes how to install the ZeBu software and hardware.
- [2] The *ZeBu-Server Compilation Manual* describes the compilation process.
- [3] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.
- [4] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.
- [8] The *ZeBu zRun Emulation Interface Manual* describes the **zRun** emulation control interface and how to use the different functions.
- [10] The *ZeBu-Server Smart Z-ICE Manual* provides detailed information on how to configure and connect the Smart Z-ICE interface to an external system.
- [11] The *ZeBu-Server Direct ICE Manual* provides detailed information on how to configure and connect a Direct ICE module for the connection of emulated DUT I/O pins to a target system and hard cores.
- [13] 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.
- [14] 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.
- [15] 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.



11 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 2-bis, Voie La Cardon Parc Gutenberg, Bâtiment 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 95054 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	EVE-Taiwan Branch 5F-2, No. 229, Fuxing 2nd Rd. Zhubei City, Hsinchu County 30271 TAIWAN (R.O.C.) Tel: +886-(3)-657-7626